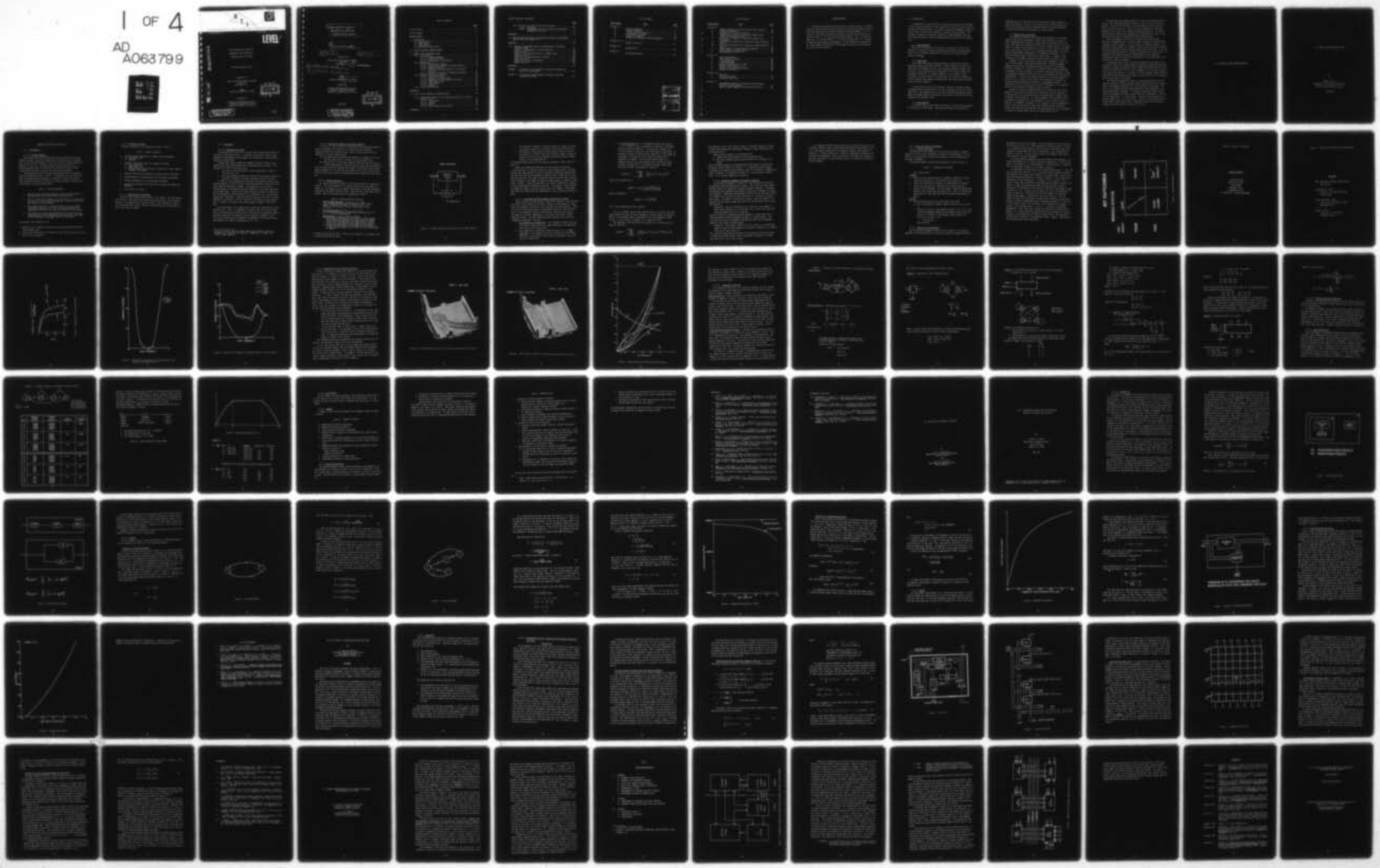
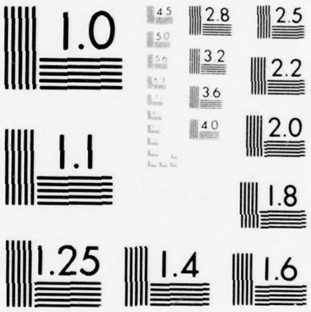
D-A063 799 RESEARCH TRIANGLE INST RESEARCH TRIANGLE PARK NC SYST--ETC F/G 9/2
BASIC RESEARCH IN SUPPORT OF CONCURRENT FAULT MONITORING IN MOD--ETC(U)
JUN 78 J W GAULT, P N MARINOS, K S TRIVEDI N00039-77-C-0363
INCLASSIFIED RTI/1504/00-01-1 NL

1 OF 4
AD
A063799





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

RESEARCH TRIANGLE INSTITUTE

RTI/1504/00-01 I

June 1978

BASIC RESEARCH IN SUPPORT OF CONCURRENT FAULT MONITORING IN MODULAR DIGITAL SYSTEMS

Interim Technical Report

Prepared for

Naval Electronics Systems Command
Code 304
Washington, D.C.

Under
Contract No. N00039-77-C-0363

Prepared by

Systems and Measurements Division
Research Triangle Institute
Research Triangle Park, NC 27709

RESEARCH TRIANGLE PARK, NORTH CAROLINA 27709

6

BASIC RESEARCH IN SUPPORT OF CONCURRENT FAULT MONITORING IN MODULAR DIGITAL SYSTEMS.

9 Interim Technical Report,

11 Jun 78

12 364 p.

Prepared for

Naval Electronics Systems Command
Code 304
Washington, D.C.

14 RTI/15Ø4/ØØ-Ø1-1

10 J.W./Gault, P.N./Marinos,
K.S./Trivedi D.L./Parnas

Under

Contract No. 15 N00039-77-C-0363

Prepared by

Systems and Measurements Division
Research Triangle Institute
Research Triangle Park, NC 27709

June 1978

411 039

TABLE OF CONTENTS

Page

TABLE OF CONTENTS (CONTINUED)

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

This report was prepared by the Research Triangle Institute in cooperation with the University of North Carolina, North Carolina State University and Duke University for the Naval Electronics Systems Command. Mr. R. D. Alberts of the Research Triangle Institute was the project Laboratory Supervisor and Mr. J. B. Clary was the Project Engineer. Dr. J. W. Gault of NCSU, Dr. P. N. Marinos and Dr. K. S. Trivedi of Duke and Dr. D. L. Parnas of UNC were the principal investigators. Sincere thanks are expressed to Mr. Larry Sumney of ELEX 304 for the guidance and support provided during the course of this contract.

1.0 INTRODUCTION

The Research Triangle Institute, along with faculty and staff members of the Triangle Universities--Duke, North Carolina State University and the University of North Carolina, is currently conducting fundamental research in the area of on-line fault detection in modular digital systems. The program is sponsored by the Naval Electronics Systems Command, Code 304. This document reports on the work accomplished during the 1977-78 academic year.

1.1 Study Objective

The overall objective of this effort is to discover on-line fault detection, isolation and repair techniques and measures of effectiveness which will be ultimately applicable to tactical and strategic modular digital systems.

1.2 Study Scope

This study specifically addresses digital system built-in-test as opposed to analog system built-in-test. Digital systems of interest are those which are composed of basic stored program computer elements. Specifically, such structures have input/output (I/O), control, memory and arithmetic processing. Of primary interest in this research is the derivation of results which will be applicable to structures composed of modular hardware and software building blocks that constitute the elements of basic digital computers.

To focus the efforts of this research, emphasis is placed on those techniques which are applicable to on-line (as opposed to off-line) fault monitoring. The exclusion of off-line fault detection approaches acknowledges the immense amount of work which has gone on in the area of off-line, automatic test equipment (ATE).

1.3 Study Approach

Within the scope of work stated in Section 1.2, the following approach is being taken in this study: Continuous and sample fault monitoring techniques which are applicable to on-line modular digital systems are being investigated. Criteria for distributing the resulting fault monitoring and reporting resources throughout modular systems hierarchies, including both hardware and software, are being derived.

1.4 Overview of Study Results

The subtasks reported in this report were identified in a previous study as areas where fundamental research could potentially lead to a better understanding of the basic issues related to modular digital computer fault detection techniques and measures of effectiveness. The importance of a modular system approach to military digital system realization lies in the ability of users of such systems to effect repair in a timely manner through module replacement. Of fundamental importance is the ability to detect module faults and to report them in a manner so that system users can readily identify the faulty member and effect repair.

Subtasks I and II specifically address some fundamental issues relevant to fault detection and handling techniques. These two subtasks, commonly referred to as Continuous Monitoring (Subtask I) and Sample Monitoring (Subtask II), explore error detection and handling techniques. The emphasis to date on Subtask I has been in identifying the basic issues of programmable computers which have to do with hardware and software fault communication. Of particular interest have been the interface questions which exist as a result of partitioning fault detection and handling resources between hardware and software in programmable machines. The investigators on this subtask view as nearly inseparable, error detection and error handling which ultimately lead to recovery.

Subtask II focuses on sample monitoring as a technique which holds promise of being able at some point to effectively utilize large-scale integrated (LSI) circuit devices such as microprocessors to detect faults in modular digital machines. The attractiveness of this approach lies in the potential universality of the approach to a wide variety of digital modules. A fundamental assumption in this subtask is that programmable LSI devices which may be used for sample fault monitoring must operate at slower rates than the processes which are being monitored--as a consequence, to reduce the computational load of the sample monitor itself.

One technique which appears promising is the use of statistical monitoring. This technique has been successfully used in off-line testing. However, in the off-line situation, one has complete control over the inputs to the unit under test. Herein lies the consequential difference between off-line and on-line statistical sample monitoring. The sample monitoring work reported here addresses the issues which are basic to systems where the inputs are unspecified and non-deterministic.

The third subtask, which has been identified as an area where additional fundamental work is clearly needed to support fault monitoring in modular digital systems, is the BIT resource allocation subtask. The objective of this subtask is to identify ways to effectively distribute BIT facilities throughout various modular system hierarchical levels, including individual module collections of modules (subsystem) and system levels. To accomplish this objective one must understand both fault detection techniques and ways to assess the effectiveness of such techniques at the hierarchical levels of interest.

The approach taken to this subtask has emphasized both analysis and simulation as means for identifying and assessing BIT approaches and effectiveness at various hierarchical levels. Models previously used have been found to be effective for off-line and inadequate for on-line fault detection technique evaluation. For example, in the evaluation models presently being used, error latency and the impact of faulty on-line error detectors have not been totally considered.

The following subsections present a detailed description of the BIT research subtasks, including subtask problem definitions and the progress made on each subtask during the first two semesters of the study. The section is divided into three major subsections which correspond to the three University/RTI subtasks. It should be noted that these sections reflect the emphasis and style of the individual investigators. Section 2.1 was authored by J. W. Gault (NCSU). Section 2.2 was written by P. N. Marinos (Duke) and K. S. Trevidi (Duke). Section 2.3 was written by D. L. Parnas (UNC) and Don Bowles (UNC). Appendices A and B are two masters theses which have resulted from this work so far.

2.0 DETAILED SUBTASK PROGRESS REPORTS

2.1 SAMPLE FAULT MONITORING (NCSU)

by

James W. Gault
Department of Electrical Engineering
North Carolina State University

June 1978

STANDARD ON-LINE FAULT MONITORING

2.1.1 Introduction

2.1.1.1 Problem Statement

The origin of the work presented here was an earlier effort at the Research Triangle Institute to investigate the feasibility of a standard built-in-test circuit suitable for use with a family of digital electronic modules [1], [2], [3]. The basic objective during the search for a standard approach to BIT is that this will make the inclusion of test circuits more naturally a part of the normal design procedure.

The present direction of the work discussed here has as a beginning point the assumptions given in Table 1. The approach to on-line fault detection presently being considered is statistical, since this seems to offer an effective method of achieving a standard approach to monitoring a wide variety of different module types.

Table 1. Problem Assumptions

1. Modular Digital electronic equipment which has been fielded, i.e., has passed design and manufacturing acceptance tests.

2. The BIT circuits will monitor on-line operations of a digital module and will not alter normal processing. The ability to insert test vectors is not considered.

3. The testing objective is to detect multiple as well as single logic faults. Although it is not clear what the response to intermittent faults will be, they are not specifically excluded.

4. The behavior of a module being monitored will be characterized, assuming stationary and independent input statistics. The impact of non-stationary and dependent inputs will be studied.

The purpose of this research is to:

1. Develop analytic methods for statistically characterizing digital electronic modules, and
2. Evaluate the effectiveness of statistical fault monitoring as an on-line built-in-test technique.

2.1.1.2 Direction of Effort

The major milestones in this project are given in Table 2.

Table 2. Project Milestones

1. Develop software capability to support the investigation:
   a. Simulation, and
   b. Plotting

2. Develop a statistical model for a module including:
   a. Input statistics,
   b. Fault model, and
   c. Network output probabilities as a function of inputs, network structure, and faults.

3. Develop a monitoring and detection strategy based on the model.

4. Develop measures of effectiveness for evaluating the approach.

5. Develop experiments to test the methods developed.

6. Evaluate the outcome of the effort and reconsider the models and approach.

7. Final report of results.

2.1.1.3 Organization of the Report

There are three sections which follow in this report. The next section will outline in some detail the related results reported in the literature. Section 3 then summarizes the current status of this research and reports the results obtained to date. Section 4 defines the plans and approach for work in the next six months.

2.1.2 Background

2.1.2.1 Deterministic Testing

The purpose of this section is to review other published work which is relevant to the present effort. In addition to providing a general background and understanding of the problem, the credibility of the statistical approach is established.

Deterministic testing procedures attempt to identify explicit test vectors which detect the presence of a predetermined set of faults. This approach falls short in two major ways:

1. The set of faults considered is often unreasonable in terms of real failures, and
2. The input sequences required to test practical networks (which are typically sequential) are often extremely difficult to derive.

In light of this, manufacturers faced with the problem of testing tremendous volumes of circuits of ever increasing complexity, resorted to using test sequences generated at random.[1] This approach calls for the resulting outputs to be compared with the response of a so-called "gold unit" to the same inputs. There are of course difficulties with this approach in that the generation and maintenance of the reference is not a trivial task. Perhaps more troublesome is the fact that the quality of the test is almost impossible to ascertain. In 1975, work [5,6,7,8] focusing on the development of a theory for probabilistic testing began to appear in the literature.

For the purposes of this paper we will limit our consideration to fault detection only. This is a reasonable limitation since the on-line monitor which is envisioned will monitor units at the level of a replaceable module and hence, detection and diagnosis may be considered synonymous. Very little work concerning fault diagnosis has appeared in the literature with the exception of a paper by Deschizeaux et al. [9].

[1] The Fluke Trendar, Data Test Corps, Data Tester Services, and Microsystems, Inc., MICRO 50U are examples of commercial IC testers which utilize random sequence.

2.1.2.2 Probabilistic Modeling of Network Behavior

In particular, work by Parker and McCluskey [5,6] developed methods by which the probabilistic behavior of a network could be described. Summarized below are the most salient results from this work which are applicabl to the work reported here.

R1.) For combinational networks the probabilistic behavior of an output may be derived as a function of input probabilities.

R2.) There exists a set of input probabilities such that the no two n-variable combinational functions have the same output probability.

Since for any particular combinational network, the possible faults simply map the network to a new function, then there exists a set of input probabilities which may be used to distinguish the good function from any faulty one.

2.1.2.3 Network Statistics

The idea of monitoring a module in an on-line situation is depicted in Figure 1. The basic objective is to gather statistics on the inputs ($\underline{x}$), states ($\underline{s}$), and outputs ($\underline{z}$) of a general module and to conclude the present condition of the module from this data. What statistics then should be collected? Some of the possibilities are enumerated in Table 3.

Table 3. Possible Network Statistics

1. <u>Ones (zeroes) Counting</u> - For $x_i \in \underline{X}$, $s_i \in \underline{S}$ and $z_i \in \underline{Z}$, count the number of occurrences of ones (zeroes) over an experiment of length N. Then prob $(x_i = 1) = \text{COUNT}_{i/N}$. The statistics are the count values for all x_i, s_i, z_i.

2. <u>Transition Counting</u> - For $x_i \in \underline{X}$, $s_i \in \underline{S}$, and $z_i \in \underline{Z}$, count the number of occurrences of transitions (0 → 1 or 1 → σ) over an experiment N.

3. <u>Vector Values</u> - For $\underline{X}$, $\underline{S}$, and $\underline{Z}$ as vectors:
   a) Collect the distribution of the inputs as vector values. This statistic may be kept as 2^m values for an m element vector or may be quantized into fewer ranges of values.
   b) Collect the distribution of the number of ones (zeroes) occurring in each vector over an experiment of length n.
   c) Collect the distribution of the number of transitions in a vector from one sample value to the next for an experiment of length n.

It should be pointed out that in order for any statistic to be useable, the following properties must hold:

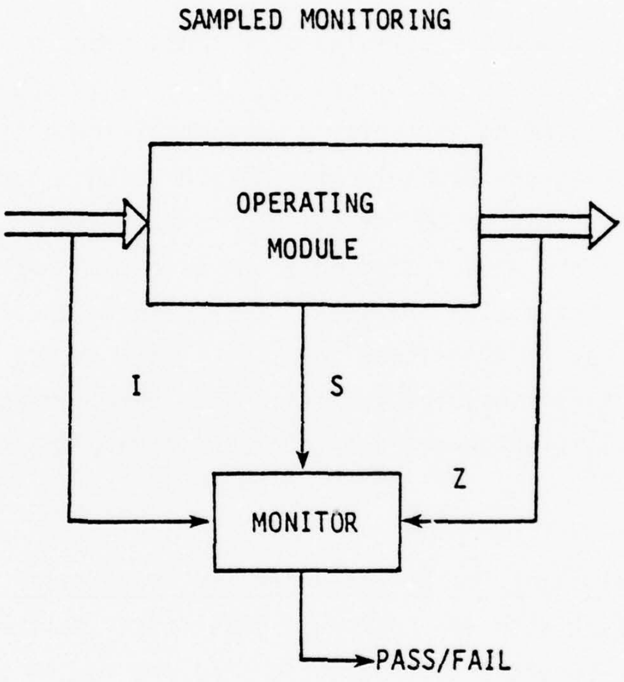


Figure 1. General Purpose On-Line Built-In-Test (OBIT) Monitor

1. The statistical behavior of network outputs or states ($\underline{Z}$, $\underline{S}$) must be derivable a priori as a function of the input statistic ($\underline{X}$).
2. The behavior of the output or state statistic must be varied in the presence of faults so that the difference between a measured statistic and its a priori expected value may be used to detect faults as in R2 stated earlier.

All reported results have used line counting procedures (items 1 and 2 of Table 3).

Parker [4] examined the usefulness of three types of statistics: 1) ones counting, 2) transition counting, and 3) edge counting. He demonstrated that edge counting is really a subcase of transition counting and that the best results can probably be obtained using a combination of ones and edge counting. This seems intuitively correct since the ones counting gives a measure of the number of inputs and edge counting, a measure of the order of inputs. Hayes [10] provides a very careful theoretical treatment of the effectiveness of transition counting. This method is often applied by commercial testers since the statistics are very compact. Hayes shows that there are faults which are undetectable if only transition counting is used.

2.1.2.4 Evaluating the Effectiveness of Statistical Testing

Often attendant with the notion of statistical testing is the idea that the easiest input sequence to be used is simply a random sequence. Hence, there are frequently references in the literature to random testing. Clearly one of the motivating notions of statistical testing is that the difficulty of test generation found in deterministic testing can be avoided. The question of primary importance is, "How good a test does a random sequence provide?" Losq [13] obtains general expressions for three very fundamental parameters. They are:

1. The probability of escape [ES] - This parameter defines the probability that a faulty unit will escape detection in an experiment, given that a failure does exist.
2. The probability of rejecting a fault-free unit, i.e., a false alarm [FA] - This parameter defines the probability that a fault-free unit will be found in error after an experiment, given that the unit is fault-free.

3. The test stringency [ε] - This parameter defines the window (Z -ε, Z + ε) which is used in accepting or rejecting a unit based on some statistical measure Z. Losq considers testing in an off-lin situation so that the input values may be controlled. An experiment consists of measuring the statistics of the output and the input (or controlling it to a specified value) over a sequence of N samples. For any given network there is an expected value of the output statistic for a particular input statistic, denoted $E_z(x)$. The probability that a fault-free unit does not pass an experiment is given by:

$$\text{prob(FA)} = 1 - \sum_{k = E_z(x) - \varepsilon}^{E_z(x) + \varepsilon} \binom{N}{k} \cdot E_z(x)^k \cdot [1-E_z(x)]^{N-k}$$

which can be simplified to

$$\text{prob(FA)} = \varepsilon\text{rfc}\left(\varepsilon \sqrt{\frac{N}{2 \cdot E_z(x) \cdot (1-E_z(x))}}\right)$$

which is bounded by

$$\text{prob(FA)} \leq \varepsilon\text{rfc}\left(\varepsilon\sqrt{2N}\right)$$

εrfc is the complementary error function.

This is a dramatic result which indicates that in the limit, the false alarms depend only upon the test stringency ε and the length of the experiment N. As one might suspect, the penalty for a wide acceptance window (large ε) is an increase in escapes, i.e., higher prob (ES).

The computation and description of the prob (ES) is not as direct as that for prob (FA).

$$\text{prob(ES)} = \sum_{k = E_z(x) - \varepsilon}^{E_z(x) + \varepsilon} \int_0^1 \binom{N}{k} \cdot E_z(x)^k \cdot (1-E_z(x))^{N-k} \cdot \emptyset(E_z(x))dEz$$

The complexity arises from Ø $[E_z(x)]$ which is a density function of faulty circuits which produce the same output statistic as that expected for the fault-free circuit.

Implied by this function is the necessity to:

1. Identify the faults which are to be considered, and
2. Derive the network output statistics under the influence of all faults.

One of the major problems in evaluating the effectiveness of statistical testing is the derivation of Ø $[E_z(x)]$. It is possible, albeit tedious, to compute Ø for combinational networks, since it is possible to compute $E_{(x)}$. Since, in general, for sequential networks, it is not possible to define $E_z(x)$, it is likewise impossible to obtain Ø.

2.1.2.5 Statistical Methods for Sequential Networks

The testing of sequential circuits is a significantly more difficult problem than treating combinational network. Very little has been presented about statistical methods for treating sequential networks. Two papers have specifically addressed the problems of statistical treatment of sequential networks. Shedletsky and McCluskey [14] focus on the idea that a fault in a network will occur prior to any indication of a failure at an observable output. This delay is called error latency. When the network is sequential and the inputs random, then this error latency may be quite large. More formally:

Definition: The error latency ELk for a fault Fk is the number of input vectors applied to a circuit while Fk is active until the first incorrect output due to Fk is observed.

Since the latency is dependent upon the sequence of inputs used, ELk is defined probabilistically assuming the input is a random process. The basic definition may be extended to develop the notion of an acceptable detection test length of random inputs.

Definition: The latency interval n(c)k of a fault Fk is the minimum number of applied inputs necessary to achieve a probability C of observing an error due to fault Fk.

This reference described the definition of an ELM model which may be used to establish the test length n which is required to establish a minimum test quality of c over a set of fault.

On a somewhat different track and more as an extension to work in combinational networks, Parker and McCluskey [18] describe techniques for deriving the output probabilities of sequential networks using regular expressions. A regular expression is a precise, unambiguous language for describing finite automata. While no pretense is made that this approach is generally applicable, it does indicate a sense of direction for sequential circuit analysis.

2.1.3 Status and Results of Research

2.1.3.1 Problem Definition

The objective of this research is to define and evaluate a standard approach to on-line fault detection using built-in-test elements. This objective is now focused on a statistical approach to on-line monitoring as described in Chapter 2.

The standard testing strategy being developed is given in Table 4.

Table 4. A Standard BIT Strategy

For a given module

A PRIORI

1. Define a set of monitor points. In general, these will include inputs ($\underline{X}$), outputs ($\underline{Z}$), and internal state values ($\underline{S}$).
2. Derive for the states and outputs to be monitored, an expected value $E_s(x)$ and $E_z(x)$ as a function of the input statistic.
3. Derive a test stringency ε and test length n based on desired test quality, false alarm rate, and escape rate. This will involve the definition of a fault model and the fault density functions Ø (X, Ø(X,S).

ON-LINE

4. During system execution, the fault monitor must then:
   a. For an experiment of length n gather statistics on $\underline{X}$, $\underline{S}$, and $\underline{Z}$.
   b. Indicate a failure if the measured statistic (e.g., mz) is outside the acceptance window for the expected value of this statistic as a function of the measured input statistic mx.

      FAIL = $mz \geq E_z\ (mx) + \varepsilon$ or

      $mz \leq E_z\ (mx) - \varepsilon$

      PASS otherwise

2.1.3.1 Measures of Effectiveness

One of the most fundamental issues in this research is to establish measures of effectiveness which can be used to accept or regret statistical monitoring as an on-line strategy. The primary measures to be considered were given in Section 2.4. These measures are shown pictorially in Figure 2. In addition to test quality, false alarm rate, and escape, the error latency of an experiment will also be considered. These parameters are defined in Table 5. The more classic views of BIT measures are given in Tabl 6 and will also be considered.

For any particular experiment, once the length n and stringency ε are known, the probability of a false alarm may be computed as it does not depend (in the bound) on run time statistics. A plot of P(FA) as a function of n for two typical values of ε is given in Figure 3.

The evaluation of the probability of escape depends upon N, ε, $\emptyset(x)$ and the run time input statistic. For a particular simple sequential network, the probability of escape, which results if the occurrences of state B are used as the monitored statistic, is shown in Figure 4. This figure may be interpreted in the following way. For some N, ε ($N = 10{,}000$, $\varepsilon = .01$) values, if the input statistics are measured and the number of entries into state B is taken as the measured output, then the probability that a failure will go undetected is very sharply a function of the measured input statistic mx. If, in Figure 4, mx is around 0.5, then the probability of escape is quite low. The sensitivity of this parameter to experiment length and stringency is indicated by the plot of Figure 5. For the example circuit, which converges to steady state statistical values quite quickly, an order of magnitude change in experiment length ($N = 1{,}000$ to $N = 10{,}000$) makes little or no difference in the probability of escape. A change in the stringency from $\varepsilon = .05$ to $\varepsilon = .01$ causes a significant improvement. Recall that such a change in stringency will cause p(FA) to degrade. It is clear then that the selection of a stringency will cause a trade-off in the probability of escape vs false alarms. Both parameters improve with an increase in the experiment length. However, there may be a cost associated with long experiments.

BIT OUTCOMES

	MODULE STATUS	
TEST RESULT	GOOD	FAULTY
PASS	OPERATIONAL	ESCAPE
FAIL	FALSE ALARM	TEST QUALITY

Figure 2. Evaluation Parameters

Table 5. Typical Bit Measures

EXAMPLE MEASURES:

% FUNCTION TESTED

CYCLES FOR TESTING

MTBF CHANGE

CONFIDENCE LEVEL

% HARDWARE FOR BIT

BIT FORM, FIT, & POWER REQUIREMENT

Table 6. Probabilistic Measures of OBIT Approaches

MEASURES:

MTDF (Mean Error Latency, Sampling Ratio,
Test Quality)

Test Quality: The P
A Test (N) Will Detect A Failure
If One Exists

False Alarm Rate: The P
A Test (N) Will Detect A Failure
When None Exists

Escape: The P
A Test (N) Will Pass When
A Failure Exists

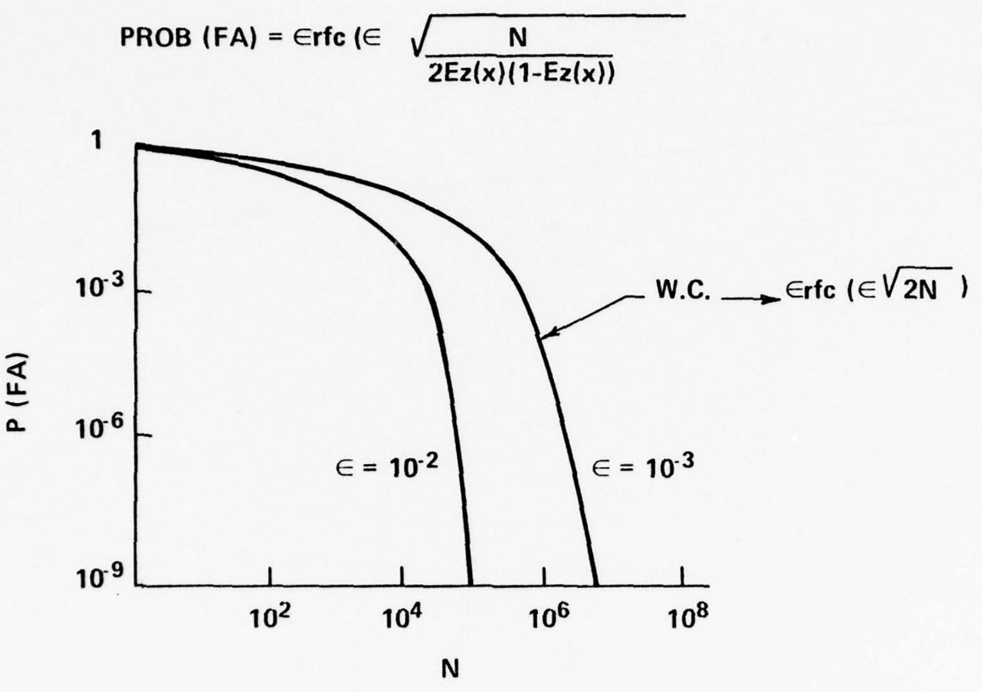


Figure 3. Probability of a False Alarm [13]

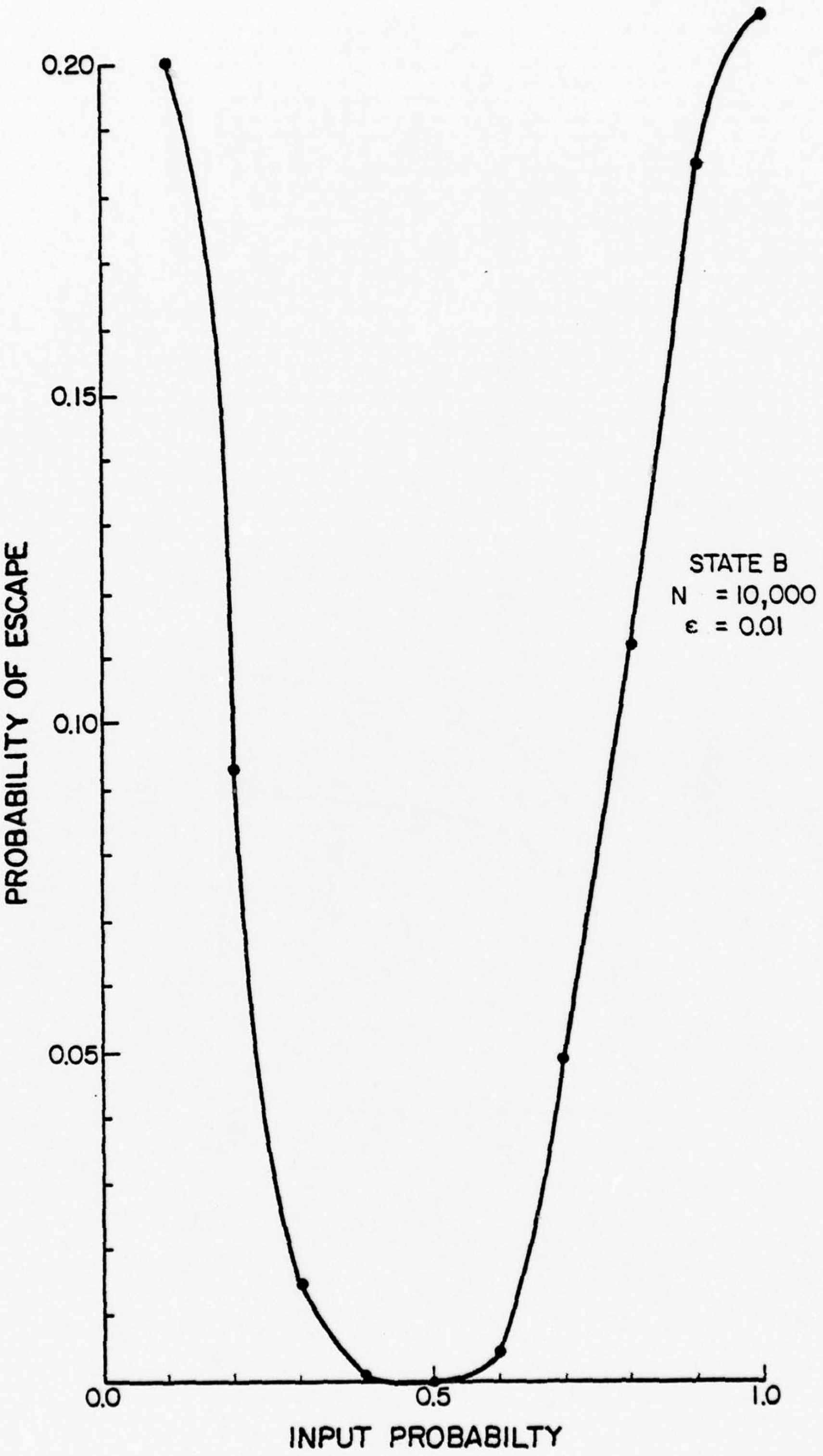


Figure 4. Probability of Escape for an Example Network and a Particular Monitored Statistic

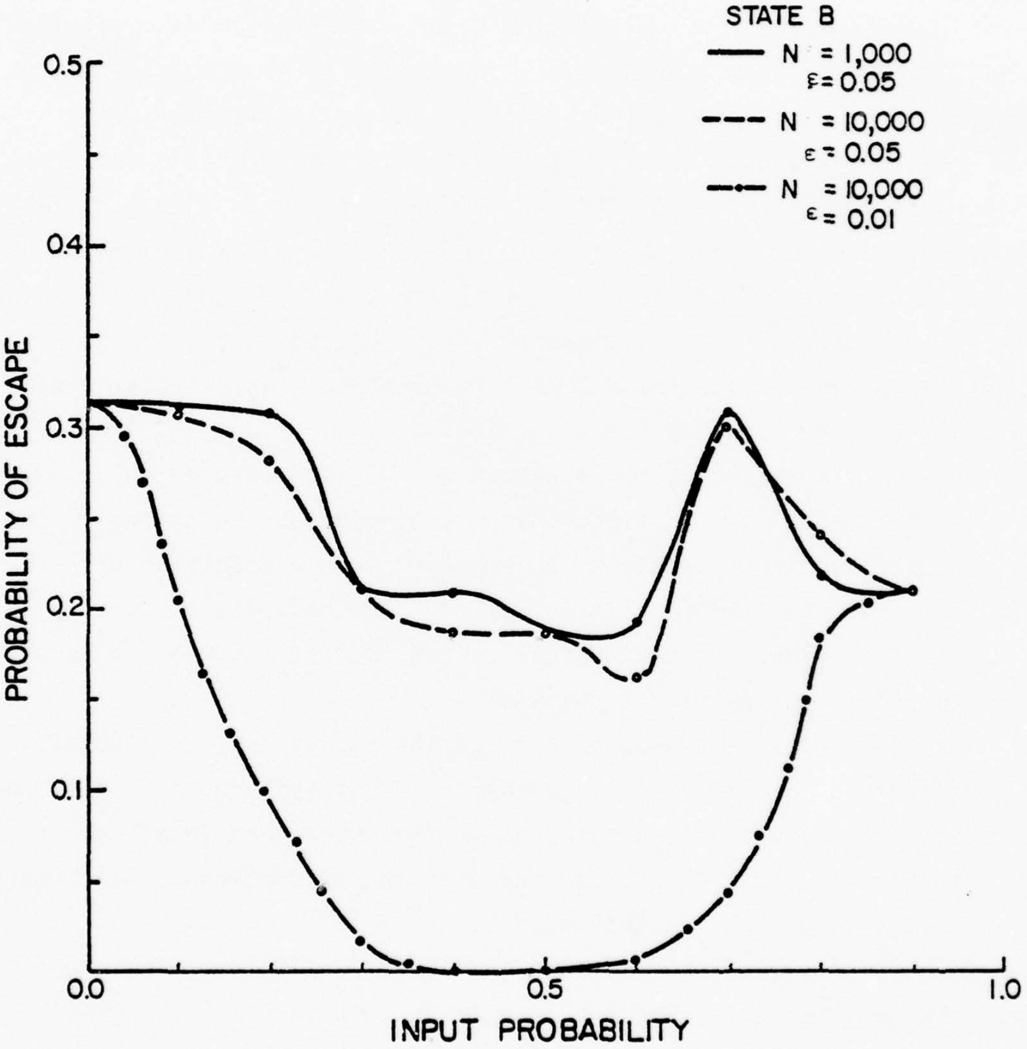


Figure 5. Sensitivity to Changes in Experiment Length N and Stringency ε

2.1.3.3 Computation of Fault Density Functions

In order to be able to compute the probability of escape and hence the test quality, it is essential to be able to derive the fault density function for a module. As stated in Section 2.4, this is one of the primary difficulties and hence objectives of this research. A fault density function describes, for a particular statistic, the number of faults which produce a particular value of output statistic for a particular value of input statistic. Mountain peaks represent places where a large number of faults produce identical behavior as far as this particular statistic is concerned. Figures 6a and 6b show fault density functions for two output statistics. If a small value of the input were measured in an experiment and a small value of the output statistic (say state B of Figure 6a), then, since a large number of fault functions (high density) are present in this area, it will be difficult to distinguish which function produced the result. Since we can compute the probability of state B as a function of x for the fault-free network, we can determine:

1. If the fault-free value is within ± ε of the measured value, then either the module is functioning as a good machine or one of the faulty functions in the neighborhood. Such a condition will result in a high probability of escape for these faulty functions. Note that there may be another output statistic which can be used to refine the pass/fail decision.
2. If the fault-free value is outside the ± ε window, then we will declare a failure. The likelihood of this failure indication being a false alarm is a function of the experiment length and ± ε size. It should be pointed out that the assumption of stationary inputs is central to this result.

The foregoing discussion has shown the fault density function and its utility. The question at hand is how can it be computed!

If, for a particular statistic Z, we can compute the output probability Z(x) for the good network, then the network function may be perturbed by a fault f and $Z_f(x)$ computed. This is done in Figure 7 for an example circuit with 9 assumed faults. The number of failures producing a particular output, quantized to some practical size (.05 was used in Figure 6), may be accumulated and plotted as a third dimension.

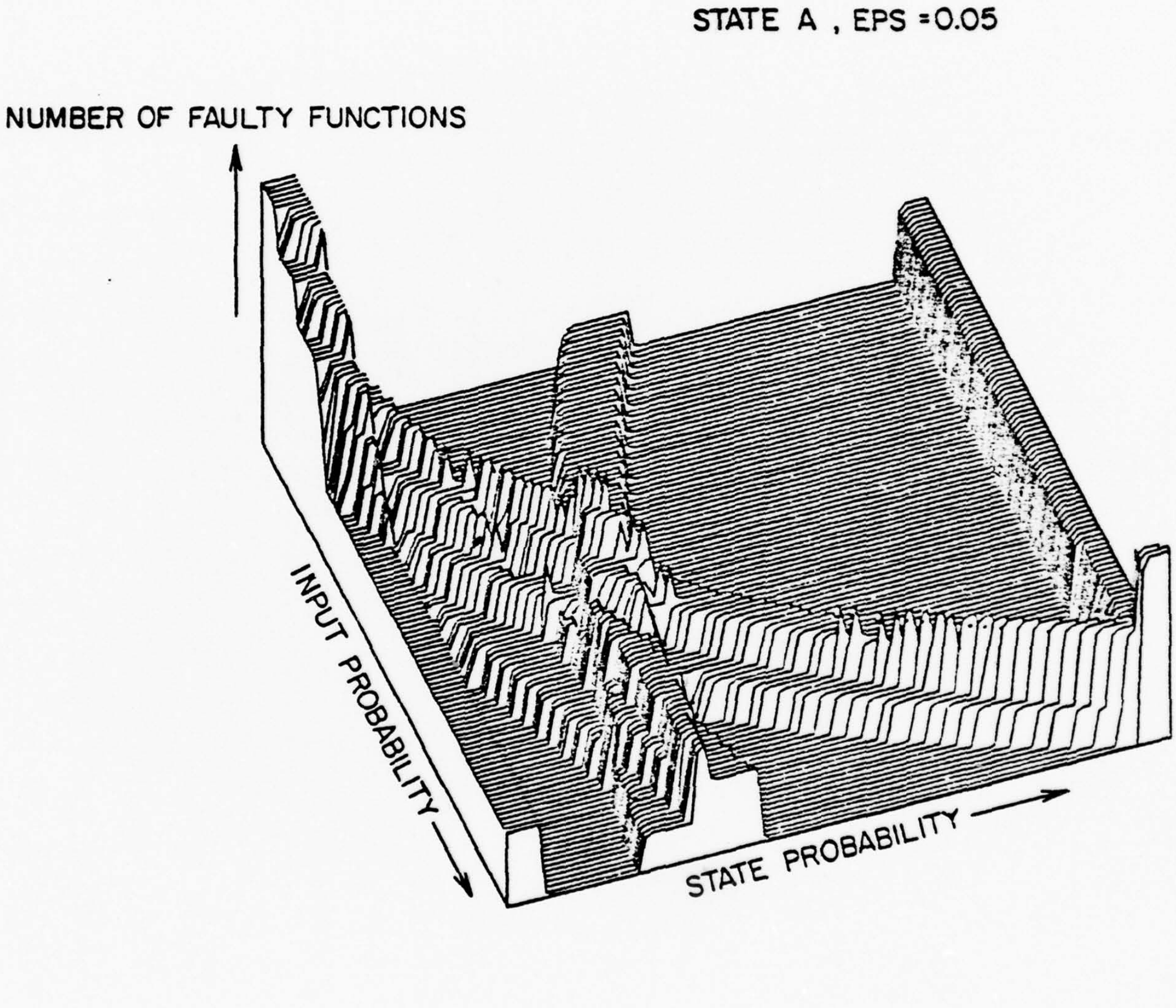


Figure 6a. Fault Density Function for a Particular Output Statistic State A

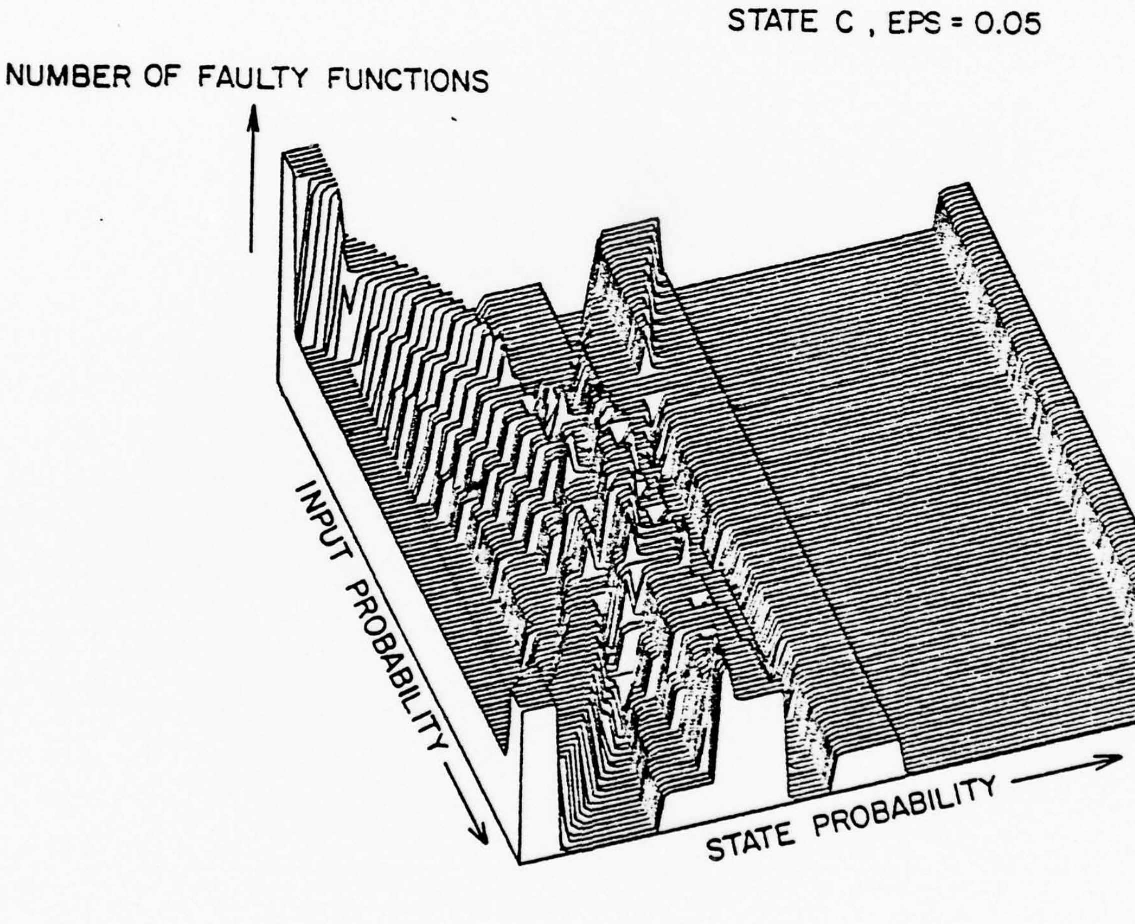


Figure 6b. Fault Density Function for a Particular Output Statistic State C

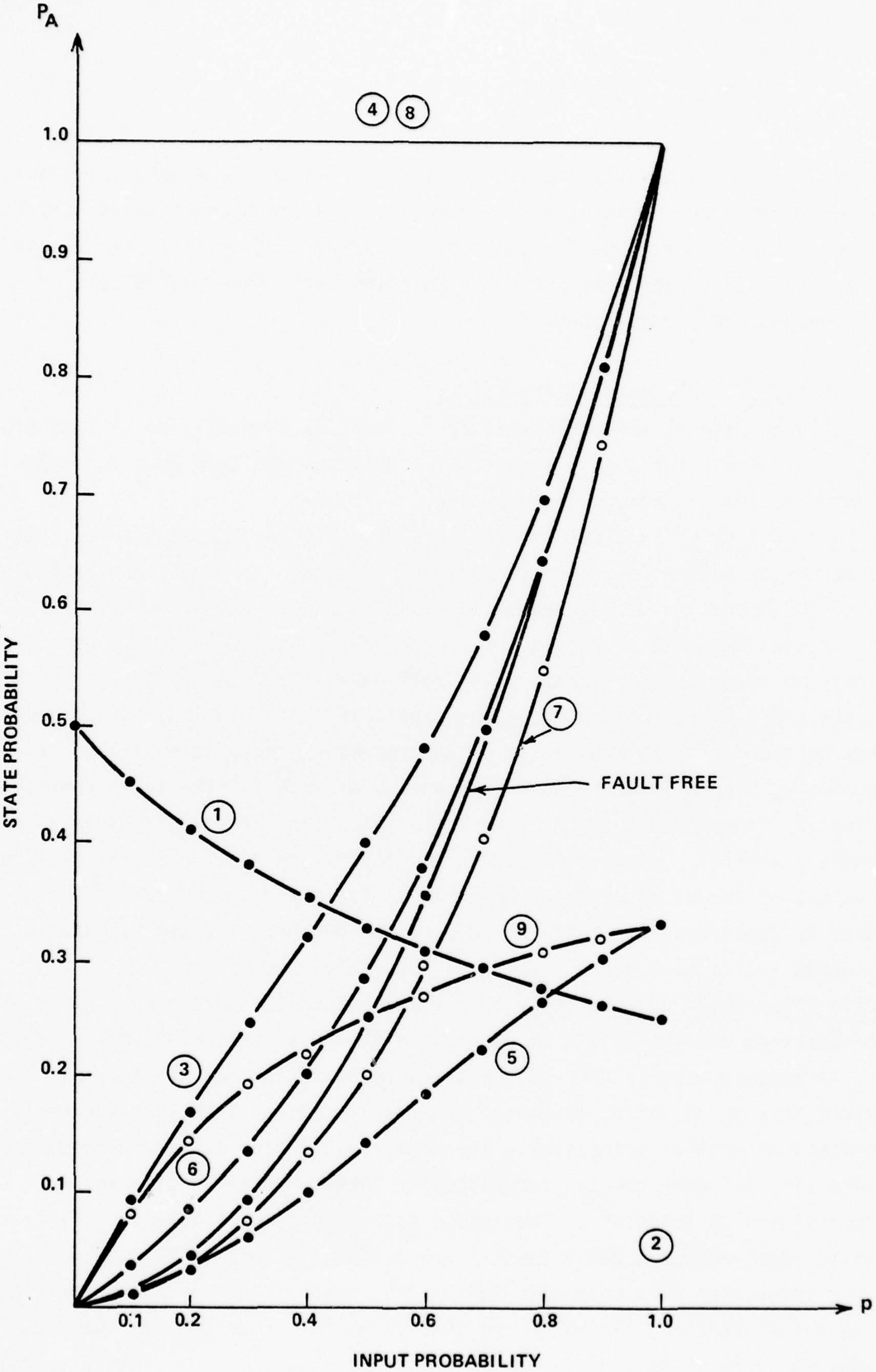


Figure 7. Output Behavior For Good And Faulty Networks

This approach is clearly impractical for any reasonable size problem and other methods are being studied. However, as a starting point we will continue to develop methods for describing Z(x) for sequential networks and then use Z(x) to generate Ø(z) by simulation until more analytical approaches can be developed.

2.1.3.3.1 Sequential Primitives

In an attempt to derive z(x) for sequential networks, we will consider the derivations for simple sequential primitives and then look for ways to form more complex component functions.

Simple Controllers - Simple controllers are defined by state tables, state diagrams or sequential networks from which a state table may be derived.

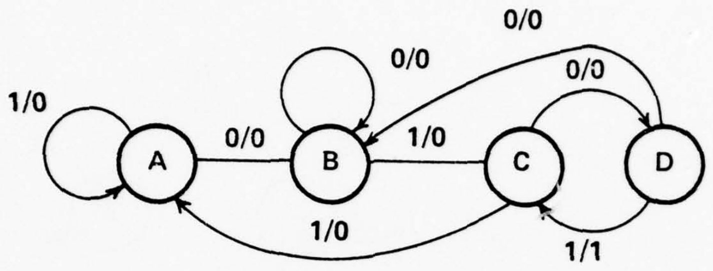
The operation of the sequential machine under random inputs may be treated as a Markov process since the present state depends only upon the previous state and the input. A transition matrix P may be formed from the state table (diagram) and the state probability after a long set of inputs may be found from P^n with $n \to \infty$. An example of these calculations for a simple, single input network is shown as example 1. The fault density function can be found for a network by evaluating the state diagram < and hence P matrix), which results with the occurrence of each fault, and then evaluating the output/state probabilities for the new machine. The fault density functions for states A and C of the network of example 1 and 10 assumed faults were given earlier as Figures 6a and 6b.

Flip-Flops/Shift Registers/Counters - The method for obtaining Ø given in conjunction with example 1 is applicable to any sequential circuit. Since it is computationally unfeasible, a new approach is considered here. The basic idea is to define the output and fault density functions for well defined sequential primitives. The objective is then to derive these same functions for more complex composites of these primitives by some composition of functions. The output functions for a flip-flop (FF) 2-bit shift register and a 3-bit counter are derived below.

The notion of a composite function is demonstrated by first deriving the output function for an FF and then using this result to derive the behavior of a shift register which is comprised of FF's. A second approach is then used; the shift register output function is derived directly; and

Example 1. Computation of Output Probability for Sequential Machines

State diagram:



Transition matrix: probability of an input =1 is P, 0 is 1-p

$$P = \begin{array}{c} \\ \\ \end{array}\begin{array}{ccccc} & A & B & C & D \end{array}$$

$$P = \left[\begin{array}{ccccc} A & p & (1-p) & 0 & 0 \\ B & o & (1-p) & p & 0 \\ C & p & 0 & 0 & (1-p) \\ D & 0 & (1-p) & p & 0 \end{array}\right]$$

$p^n{}_{n\to\infty}$	p^2	$1-2p+2p^2-p^3$	$p(1-p)$	$(1-p)^2p$
p^n converges for	"	"	"	"
	"	"	"	"
n=3	"	"	"	"

If states are taken as outputs the probability of the state outputs as a function of the input probability is given by

probability of state $A=P_A=p^2$

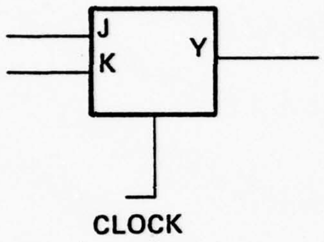
Simili $P_B=1-2p+2p^2-p^3$

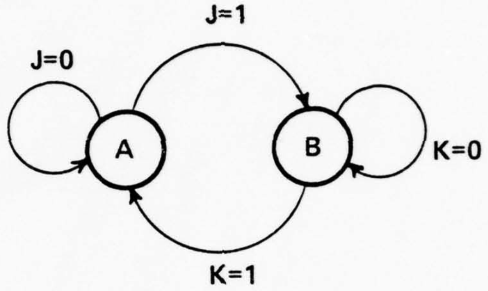
$P_C=p(1-p)$

$P_D=p(1-p)^2$

the results of the two approaches are shown to agree.

Example 2 - Derivation of an FF Output Function





$j \triangleq$ prob $J=1$

$k \triangleq$ prob $K=1$

$j+k \geqslant 1$

$p(A)+p(B)=1$

$$P = \begin{bmatrix} 1-j & j \\ k & 1-k \end{bmatrix}$$

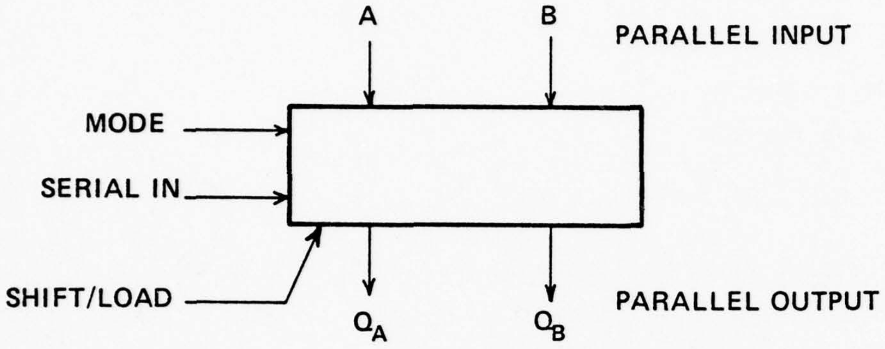
$$p(A) = \frac{k}{j+k} \qquad p(B) = \frac{j}{j+k}$$

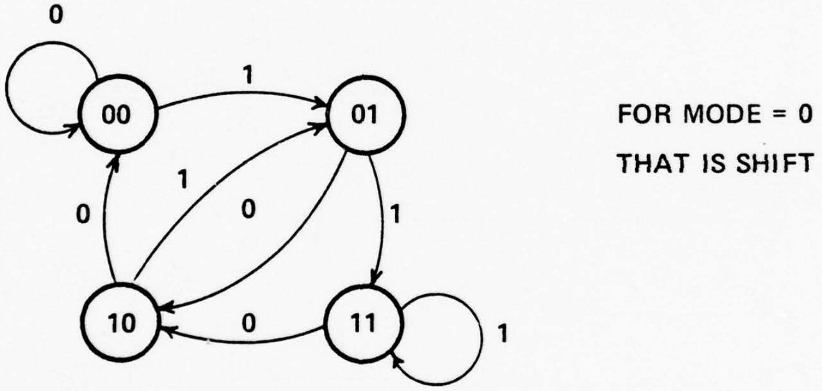
Note: a similar result may be obtained by solving the simultaneous equations which can be directly written from the state diagram.

$$p(A) = p(A)\ (1-j) + p(B)\ k$$
$$p(B) = p(A)\ j + p(B)\ (1-k)$$
$$p(A) + p(B) = 1$$

Example 3 - Two Alternative Derivations for a 2-BIT Shift Register

a. Modeled as a shift register





Statistics which might be collected are:

1. The occurrence of activity on a single output A or B (ones, transitions), or
2. The occurrence of a vector value (weight, transitions, value).

The Markov model lends itself most naturally to the vector value statistic (namely, the occurrence of states)

	Code	
State	Q_B	Q_A
A	0	0
B	0	1
C	1	1
D	1	0

The equations, taking into account parallel loading

m = mode probability, s = serial input

a, b = parallel input probabilities

$$p(A) = \bar{s}\ \bar{m} - s\ \bar{s}\ \bar{m}\ 2 - \bar{s}\ a\ m\ \bar{m} + \bar{b}\ \bar{a}\ m$$

$$p(B) = \bar{s}\ \bar{m} - s^2\ \bar{m}^2 - s\ a\ m\ \bar{m} + \bar{b}\ a\ m$$

$$p(C) = s^2\ \bar{m}^2 + a\ s\ m\ \bar{m} + m\ a\ b$$

$$p(D) = \bar{m}^2\ s\ \bar{s} + \bar{m}\ m\ \bar{s}\ a + b\ \bar{a}\ m$$

where $\bar{x} = (1-x)$

Statistics for the individual flip-flop outputs, qA = prob $QA = 1$, qB = prob $QB = 1$ may be derived using

$$qA = PB + PC$$

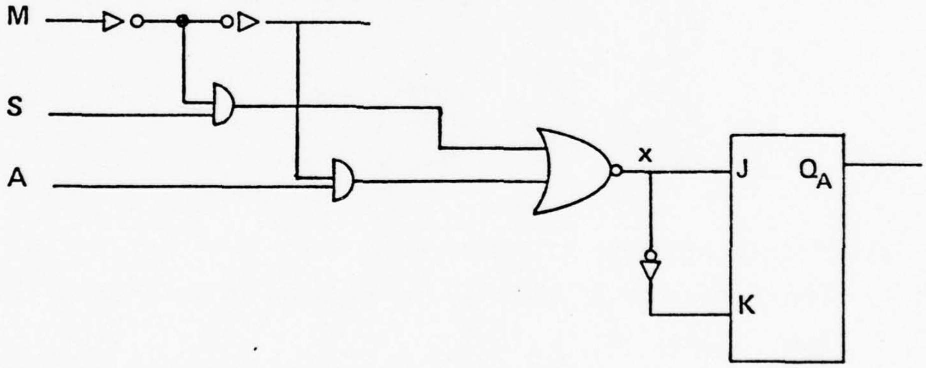
$$qB = PC + PD$$

resulting in the equations

$$qA = s\ \bar{m} + m\ a$$

$$qB = \bar{m}^2\ s + a\ m\ \bar{m} + m\ b$$

b. Modeled as a composite network



The derivation involves finding the probability of $QA = 1$ (QB is similar) by combining the FF behavior with the combinational network output X. The FF equation derived earlier can be simplified since $J = K = x$, hence

$$p(Q_A) = \frac{(1-x)}{x + (1-x)} \Rightarrow q_A = 1-x$$

Now, for the combinational network, the output probability as a function of m, p, d is

$$x = 1 - [(1-m)s + ma - ((1-m)sma)]$$
$$x = 1 - s + sm - ma$$
$$qA = s - sm + ma = s\bar{m} + ma$$

Similarly,

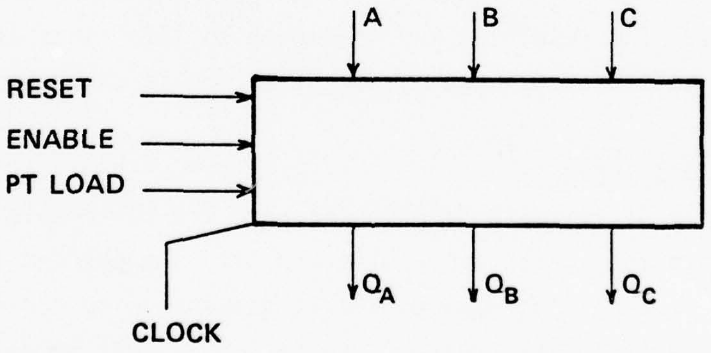
$$qB = qA - qAm + ma \text{ or}$$
$$qB = \bar{m}^2 s + am\bar{m} + mb$$

These results agree with the derivation in part A. The state probabilities can be obtained from the equations.

$$P(A) = \overline{qA} \cdot \overline{qB} \qquad p(C) = qA \cdot \overline{qB}$$
$$p(B) = \overline{qA} \cdot qB \qquad p(D) = qA \cdot qB$$

Example 3 demonstrates the composition procedure. The important implication is that once derived, a function may then be used in more complex situations. In this manner a catalog of functions and their statistical behavior may be used to derive general modules just as packages of a logic family are used to realize the modules. Example 4 demonstrates the derivation of the statistical profile for the 3-bit counter.

Example 4 - Statistical Model for a Counter



Defining the various modes as

R = low, reset $\quad \alpha = \overline{PT} \cdot L \cdot R \quad \lambda = \dfrac{\beta}{1-2}$

PT = high, count enable $\quad \beta = PT \cdot L \cdot R$

L = low, parallel load $\quad \gamma = \overline{L} \cdot R$

results in state values of

$$S_o = \frac{1 - \frac{\alpha}{1 - \gamma}\left(\frac{1}{1 - \lambda}\right)\sum_{k=1}^{N-1} ik\ (1 - \lambda^{N-k+1})}{1 + \sum_{N-1}^{N} \lambda^n}$$

$$S_j = \lambda^n\ So + \frac{\alpha}{1-2} \sum_{k=1}^{N} \lambda^{n-k}\ ik$$

2.1.3.3.2 Simulation and Experimentation

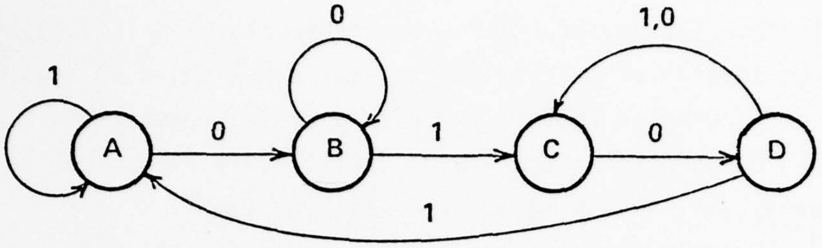
Simulation and experimentation programs have been written and run to drive fault density functions for some simple circuits with a limited number of assumed faults.

The network of example 1 was simulated with ten single stuck faults and the results plotted. These plots are given as Figures 6a and 6b. In addition, simulation experiments were run to verify the state probability results made analytically. Several examples were run showing very good results, i.e., differences of less than ± .06 between simulated and experimental results. The results often converged to this range after as few as 100 input conditions. Example 5 shows the results of a sample run.

2.1.3.4 Input Stationarity

The effectiveness of the methods proposed here are vulnerable to the validity of the assumptions which are applied in order to perform the required analysis. Probably the most difficult assumption to verify is that of input stationarity. The truth of the matter is that we do not have any clear perception of the behavior of the inputs to a general digital module operating on-line in situations which are clearly data and application dependent. A lack of stationarity will directly impact the validity of a predicted value of an output statistic. This can be accommodated by increasing the acceptance window (± stringency ε). This, of course, results in an increased likelihood of escape and a degradation in test

Example 5: Simulation Behavior As Composed To Analytic Results



$P_A = p/(p^1-3p+3)$

$P_B = (1-p)/(p^2-3p+3)$

$P_C = (1-p)/(p^2-3p+3)$

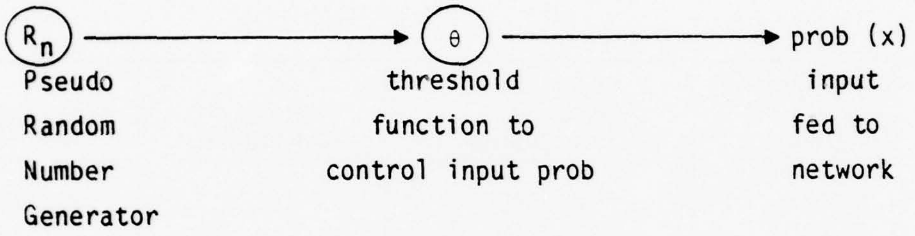
$P_D = (1-p)^2/(p^2-3p+3)$

$\lambda_1 = 1$

$\lambda_2 - \lambda_3 - \lambda_4 = (1-p)^2 p$

State	Simulated Average Visitations	Computed State Probabilities	Input Probabilities	# of Applied Vectors
A	0.0358	0.03690	0.1	10,000
B	0.3480	0.33210		
C	0.3244	0.33210		
D	0.2918	0.29889		
A	0.2815	0.28571	0.5	10,000
B	0.2814	0.28571		
C	0.2900	0.28571		
D	0.1471	0.14286		
A	0.8069	0.81081	0.9	10,000
B	0.0907	0.09009		
C	0.0921	0.09009		
D	0.0103	0.00901		
A	0.02	0.03690	0.1	100
B	0.88	0.33210		
C	0.06	0.33210		
D	0.04	0.29889		
A	0.31	0.28571	0.5	100
B	0.35	0.28571		
C	0.23	0.28571		
D	0.11	0.14286		
A	0.76	0.81081	0.9	100
B	0.11	0.09009		
C	0.11	0.09009		
D	0.02	0.00901		

quality. In order to obtain some information concerning the sensitivity of the network statistics to nonstationarity inputs, the model shown in Figure 8 was used. The function used to control the nature of the nonstationarity is given in Figure 9. The beginning and ending sections are sinusoidal and the three break times t_1, t_2, t_3 may be controlled to create the desired function. Example 6 shows two separate cases and the results. The circuit of example 1 was used.


R_n
Pseudo Random Number Generator
θ
threshold function to control input prob
prob (x) input fed to network

a. for controlled input prob (x) θ = Constant

b. for random input θ is not used.

c. for nonstationarity θ (t) is used.

Figure 8. Input Probability Control Model

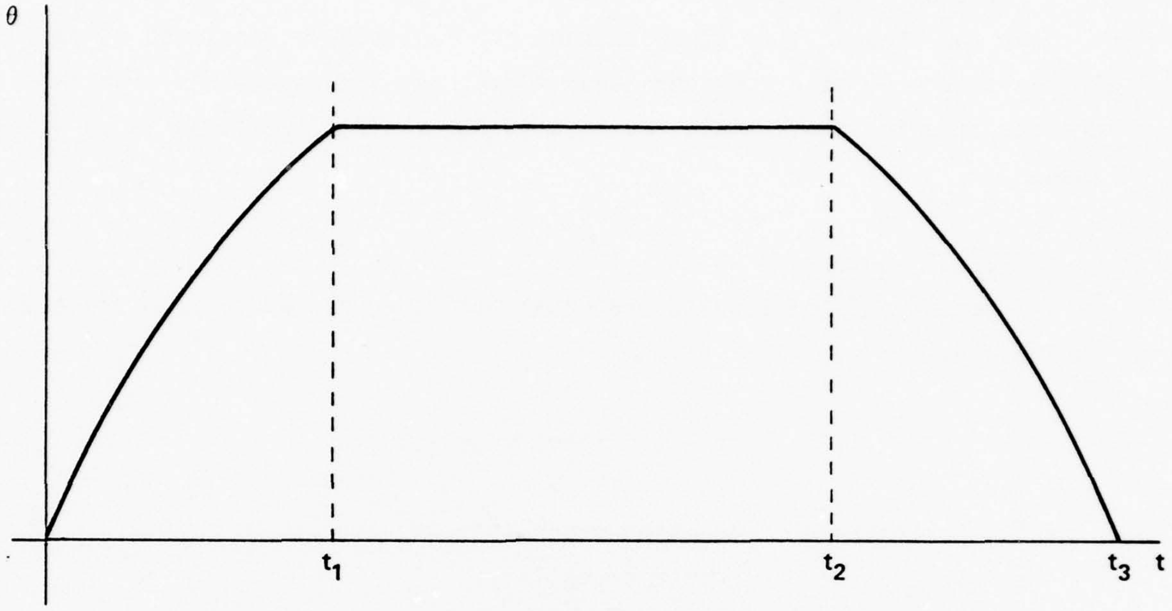


Figure 9. Non-Stationary Profile $\Theta(t)$

Example 6

A. Model: θMax = .5	Results	Theoretical*	Simulation
t_1 = 100 units	Input P(x)		.458
t_2 = 900 units	State P(A)	.209	.239
t_3 = 1000 units	State P(B)	.407	.437
	State P(C)	.248	.219
	State P(D)	.134	.105

*treating P(x) as stationary at resulting simulated value.

B. Model: θMax = .5	P(x)	--	.267
t_1 = 500	P(A)	.071	.104
t_2 = 501	P(B)	.589	.625
t_3 = 1000	P(C)	.196	.164
	P(D)	.143	.107

2.1.3.5 Fault Models

At the present time stuck package pin faults have been used as faults. In the future we will consider other models and intermittents. For the present time the stuck fault model is adequate since the focus is on other issues.

2.1.3.6 Summary

A summary of the results and capabilities developed to date are given in Table 5.

Table 5. Summary of Results

1. Demonstrated simulation capability.
2. Demonstrated plotting capability.
3. Detailed problem definition refinement.
4. Identification of measures of effectiveness with a quantitative formulation.
5. Definition of a standard approach to on-line fault monitored. Detailed steps required to characterize and test a module statistically.
6. Computation method for probabilistic output functions and fault density functions for

   Simple controllers, and
   Sequential primitives
7. Tentative definition of a fault model.
8. Experimental analysis of input stationarity.

2.1.4 Plans and Projections

The results produced to date provide considerable encouragement as to the usefulness of a statistical model in on-line fault monitoring. There are a large number of questions which can be posed as a result of the effort to date. The work which will be persued is in three major related categories:

1. Statistical characterization of general digital electronic modules.
2. Evaluation of the cost and effectiveness of on-line statistical monitoring as a standard BIT strategy.
3. Experimentation and validation of the theoretical concepts.

More detailed goals for each category are given in Table 6. The immediate emphasis is on items 1a, 2a, and 2b. These steps involve the development of the analytic tools required to model a module and evaluate the statistical approach in a tentative way. The other goals are objectives which should bring the next level of understanding and maturity to the approach.

Table 6. Research Goals

1. Statistical characterization of a module.
   a. Develop analytic methods for describing module state or output probabilities as a function of input probability. Specifically consider sequential modules.
   b. Investigate the choice of statistics which are most effective for describing a module.
   c. Study alternative strategies for pass/fail determination.
   d. Develop computationally-feasible methods for deriving experiment stringency and length for particular modules.
2. Evaluation of cost and effectiveness.
   a. Evaluate various fault models and their statistic characterization.
   b. Develop computationally feasible methods for obtaining: 1) the fault density function for a module, given the fault model from 1a, and 2) the probability of escape and false alarm as functions of experiment length and test stringency.
   c. Study the sensitivity of the approach with regard to statistical properties of the module inputs.
3. Experimentation and validation of theoretical concepts.
   a. For the present time this will involve simulation experiments designed to model modules statistically and to determine if inserted faults are detectable by a shift in statistical properties.
   b. Experiments to: 1) determine the validity of the concept for nonstationary inputs, and 2) study the performance of monitoring as a function of test stringency and experiment length.

We are very close to having the concepts developed which are required to:

1. Take a simple module and characterize it statistically, i.e., define n, e, $s(\underline{x})$, $z(\underline{x})$ for all s, z.

2. Define the theoretical performance which will be obtained for the network with an established set of faults, experiment length, (N) and stringency (ε), and
3. Demonstrate experimentally that simulated faults can be detected and that good machines are not rejected.

The development, documentation and utilization of software tools required to support this effort are an important correlary effort.

REFERENCES

1. Clary, J. B., Gault, J. W., Weikel, S. J., Whisnant, R. A., Alberts, R. D., A Study of a Standard BIT Circuit, Final Report, Contract No. 0163-76-C-0231, February 1977.

2. Gault, J. W. and Clary, J. B. "The Application of Microcomputers as On-Line Built-In-Test Elements," Proceedings of IEEE Southeastcon, April 1978.

3. Clary, J. B. and Gault, J. W., "The Use of Coding Techniques For On-Line Fault Monitoring In Microcomputer Systems," Proceedings of IEEE Southeastcon, April 1978.

4. Parker, C. P., "Compact Testing: Testing with Compressed Data," Proceeding FTCS-76, June 1976.

5. Parker, C. P. and McCluskey, E. J., "Analysis of Logic Circuits with Faults Using Input Signal Probabilities," IEEE Transactions on Computers, May 1975, pp. 573-578.

6. Parker, C. P. and McCluskey, E. J., "Probabilistic Treatment of General Combinational Networks," IEEE Transactions on Computers, June 1975, pp. 668-670.

7. Ogus, R. C., "The Probability of a Connect Output from a Combinational Circuit," IEEE Transactions on Computers, May 1975, pp. 534-544.

8. Agrawal, P. and Agrawal, V., "Probabilistic Analysis of Random Test Generation Methods for Redundant Combinational Logic Networks," IEEE Transactions on Computers, July 1975, pp. 691-695.

9. Deschizeaux, P. et al., "Statistical Fault Location in Logical Circuits," Proceeding FTCS-76, June 1976.

10. Hayes, J. P. "Transition Counts of Combinational Logic Circuits," IEEE Transactions on Computers, June 1976, pp. 613-619.

11. David, P. and Blanchet, G., "About Random Fault Detection of Combinational Networks," IEEE Transactions on Computers, June 1976, pp. 659-664.

12. Quam, J. G. and Reddy, S. M., "Referenceless Signative Testing," Proceeding of the 15th Annual Allerton Conference, September 1977.

13. Losq, J., "Referenceless Random Testing," Proceedings of the FTCS-76, June 1976.

14. Shedletsky, J. and McCluskey, E. J., "The Error Latency of a Fault in a Sequential Digital Circuit," IEEE Transactions on Computers, June 1976, pp. 655-659.

REFERENCES (Continued)

15. Ramamoorthy, C. and Han, Y., "Reliability Analysis of Systems with Concurrent Error Detection," IEEE Transactions on Computers, September 1975, pp. 868-878.

16. Stephenson, J. E. And Grason, J., "A Testability Measure for Register Transfer Level Digital Circuits," Proceedings of the FTCS-76, June 1976.

17. Maheshwari, S. N. and Hakimi, S. L., "On Models for Diagnosable Systems and Probabilistic Fault Diagnosis," IEEE Transactions on Computers, March 1976, pp. 228-236.

18. Parker, K. P. and McCluskey, E. J., "Sequential Circuit Output Probabilities from Regular Expressions," IEEE Transactions on Computers, March 1978, pp. 222-231.

2.2 BUILT-IN-TEST RESOURCE ALLOCATION

by

Dr. P. N. Marinos
Department of Electrical Engineering
Duke University

and

Dr. K. S. Trivedi
Department of Computer Science
Duke University

2.2.1 MATHEMATICAL MODELS FOR THE DESIGN AND ANALYSIS OF ON-LINE BUILT-IN-TEST*

by

Kishor S. Trivedi
Department of Computer Science
Duke University
Durham, N.C. 27706

May 1978

* Supported in part by the Naval Electronics Systems Command, Code 304, Washington, D.C., under contract RFP no. N00039-77-PR-7J-133.

2.2.1.1 Introduction

This paper is concerned with the analysis and design of on-line Built-In-Test (BIT). Such systems are characterized by on-line fault monitoring, and therefore, a study of the effectiveness of on-line fault monitors is important [1,2]. Existing models of systems analysis [3,4] are inadequate for modeling systems with on-line fault monitoring since they assume that fault detection occurs in zero time and that the fault monitor never fails. Our models will allow a finite detection latency, an imperfect fault monitor, multiple fault monitors, and multiple classes of faults.

The analysis problem occurs when the system structure is specified and we are interested in evaluating the performance of the system. Such an analysis will be probabilistic in nature since the failure modes of various components of the system are probabilistic. If a repair facility is included in our model, (i.e., the system is repairable or maintained), then the performance metric of interest is the steady state system availability. On the other hand, if the system is non-maintained (or non-repairable), the performance measure of interest is the system reliability as a function of the mission time.

Due to finite detection latency, it is possible that a fault has occurred in the system but it is not yet detected. Such a state of the system is clearly undesirable. The purpose of an on-line fault monitor is to reduce the probability that the system is in the undesirable state. We will give explicit expressions of the effectiveness of a fault monitor in achieving this goal. Our analysis will cover both maintained and nonmaintained systems.

The problem of system design is to configure the optimal system for the stated purpose. In our context, we are interested in choosing a fault monitor that yields a system with optimum cost-performance. The trade is between the cost of the monitor and the cost associated with the time the system spends in the undesirable state. In several simple cases, we will give closed form solutions that characterize the optimal fault monitor.

The basic constituent of the system that we consider is called a module as shown in Figure 1. Two types of modules will be considered. One type is the non-maintained (or non-repairable) module M and the other type is the maintained (or repairable) module M'. Module M consists of the functional unit U and its on-line fault detector D. The module M' consists of the functional unit U, the detector D and a repair facility R. An example of a functional unit is an arithmetic unit, and the corresponding detector could be its modulo-3 checker. If the functional unit is a complex processing unit, then the detector could be a software routine executed on a microprocessor. Thus, both continuous and sampled on-line fault detectors (or monitors) are modeled. If a system consists only of non- maintained modules, then it is a non-maintained system and the performance measure of interest is its reliability for a given mission time. On the other hand, for a system consisting of maintained modules, the performance measure of interest is its steady state availability [3].

Consider a series-parallel system consisting of s-serial stages where the i^{th} stage has n_i identical modules in parallel (see Figure 2). Assume that the failures of all units are independent of one another. Now if the system is non-maintained, then its reliability is given by [3]

$$R_{system}(t) = \prod_{i=1}^{s} \left[1 - (1 - R_i(t))^{n_i}\right] \tag{1}$$

where R_i is the availability of any module at the i^{th} stage.

Next, consider a similar maintained system and assume that each module has its own repair facility. Then the system availability is given by [3]

$$A_{system} = \prod_{i=1}^{s} \left[1 - (1 - A_i)^{n_i}\right] \tag{2}$$

where A_i is the availability of any module at the i^{th} stage.

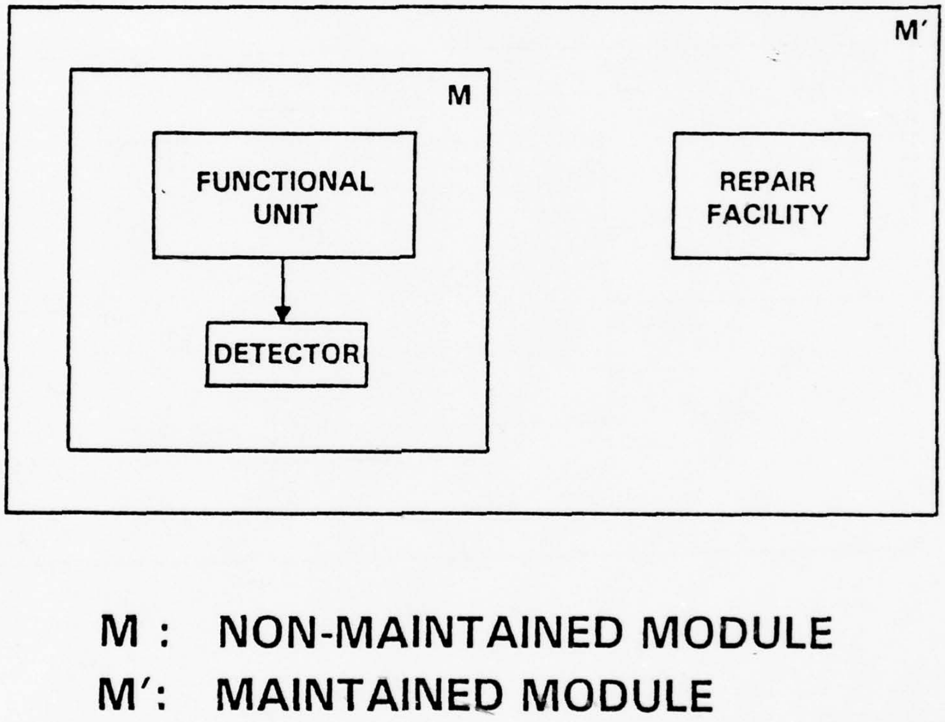
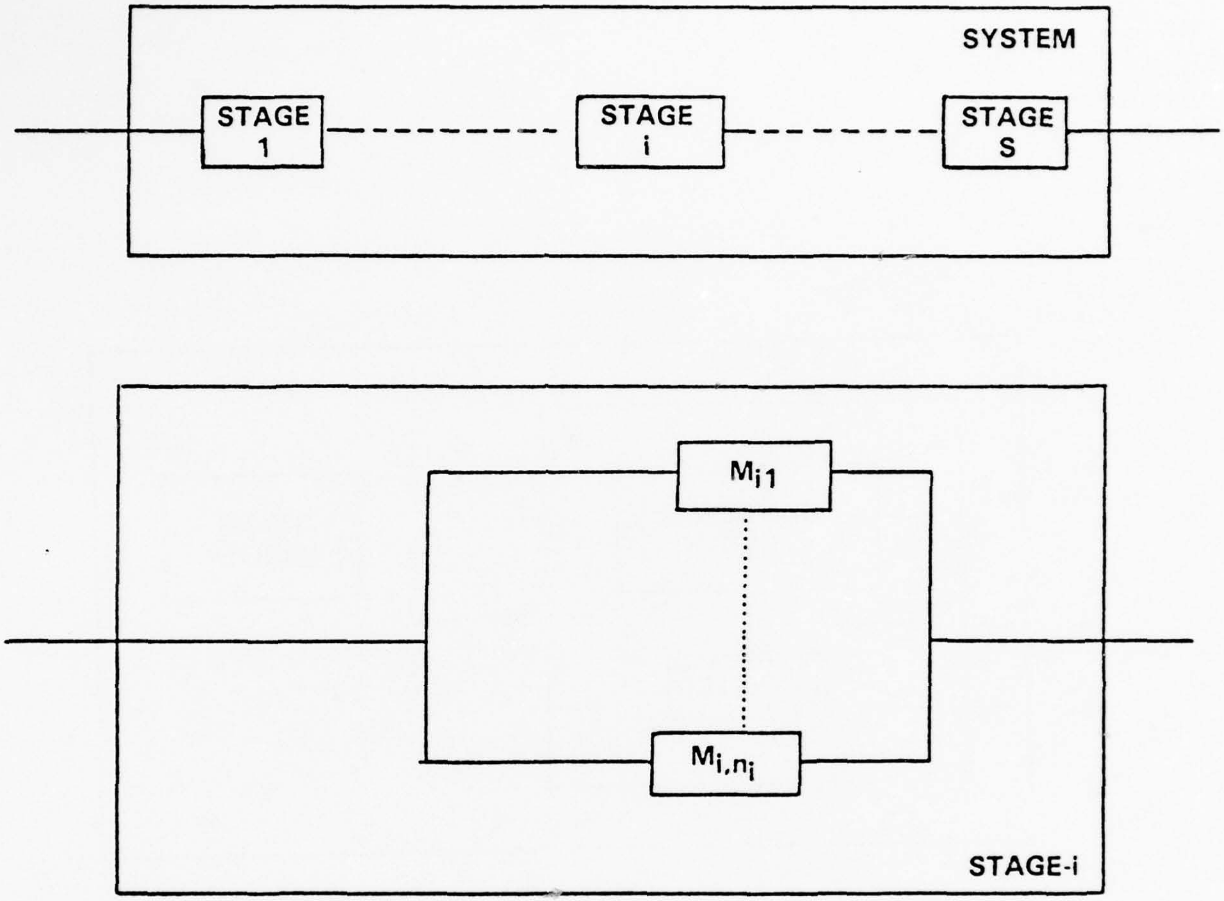


Figure 1. Basic System Module



$$R_{SYSTEM} = \prod_{i=1}^{S} \left[1 - (1 - R_i)^{n_i} \right]$$

$$A_{SYSTEM} = \prod_{i=1}^{S} \left[1 \ (1 - A_i)^{n_i} \right]$$

Figure 2. Series-Parallel Systems

From the above discussion we may conclude that for series-parallel systems of independent modules, it is enough to analyze the reliability (or availability) of individual modules. Once we have computed these, a trivial application of one of the formulae (1) or (2) given above yields the system performance measure desired. Therefore, we will only analyze the performance of module M or module M'.

Methods of analysis and design of these two types of units will be studied in Sections II and III, respectively.

2.2.1.2 Analysis

In this section, we will discuss the analysis of a maintained module, and will consider the analysis of a non-maintained module.

Analysis of a Maintained Module

We will first present the well-known analysis of a simple maintained module, devoid of the on-line detector D. Throughout the subsequent discussion we will assume that the time between two successive failures of the functional unit U is exponentially distributed with mean $1/\lambda$. Thus, the failure rate is λ and the Mean-Time-Between-Failures (MTBF) is $1/\lambda$. We assume that the time to repair is exponentially distributed with mean $1/\mu$. Thus the repair rate is μ and the Mean-Time-To-Repair (MTTR) is $1/\mu$.

The module has two possible states, F (failed) and W (working properly). The state diagram of the module is shown in Figure 3. Let P_F be the steady state probability that the module is in state F. Similarly, let P_W be the steady state probability that the system is in state W. It can be shown that [3]

$$P_W = \frac{\mu}{\lambda + \mu}$$

and

$$P_F = \frac{\mu}{\lambda + \mu}$$

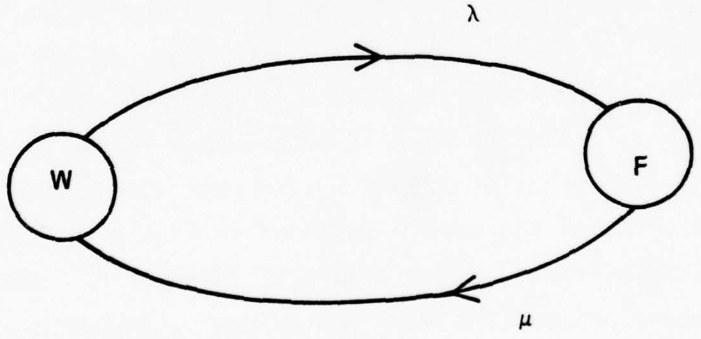


Figure 3. A Two-State Model

Now, the module availability A is simply P_W by definition. Thus

$$A = \frac{\mu}{\lambda + \mu} = \frac{1/\lambda}{1/\lambda + 1/\mu} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}} \qquad (3)$$

There are many drawbacks of this simple well known model of availability. First, it assumes that the detector is perfect; i.e, the detector never fails. Second, it assumes that the time to detect failures is negligible or the detection latency is zero. We present a model below that removes both these drawbacks.

We make all the assumptions made earlier for the simple two-state model. In addition, we assume that the time to detect failures is exponentially distributed with mean $1/\delta$. Thus the detection rate is δ and the Mean Time-To-Detect-Failures (MTDF) is $1/\delta$. The time to failure and the time to repair for the detector are exponentially distributed with mean $1/\gamma$ and $1/\beta$, respectively. The module can be in any one of the four states: W, F, D and C. In state W, the module is functioning properly; in state F, the functional unit has failed but the failure is not yet detected. In state D, the failure is detected and the functional unit is under repair. In state C, the detector has failed and it is under repair. The state diagram is given in Figure 4. The steady state probabilities for each of these states can be obtained as

$$P_W = \frac{1}{1 + \lambda/\mu + \lambda/\delta + \alpha/\beta},$$

$$P_F = \frac{\lambda/\delta}{1 + \lambda/\mu + \lambda/\delta + \alpha/\beta},$$

$$P_D = \frac{\lambda/\mu}{1 + \lambda/\mu + \lambda/\delta + \alpha/\beta} \text{ and}$$

$$P_C = \frac{\alpha/\beta}{1 + \lambda/\mu + \lambda/\delta + \alpha/\beta}.$$

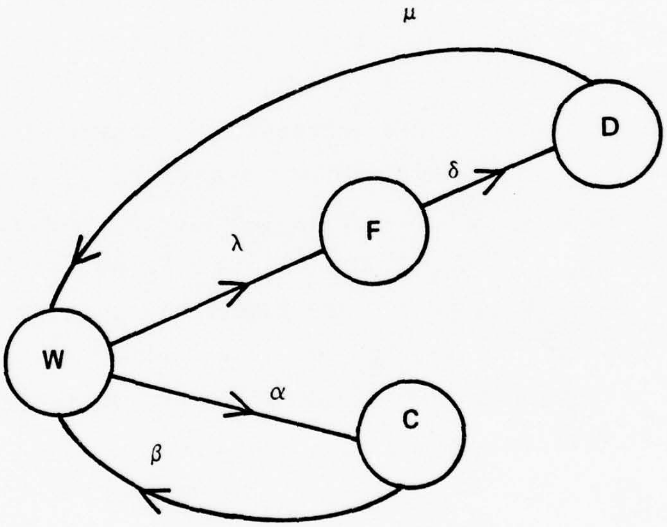


Figure 4. A Four-State Model

It is interesting to observe that when the module is in state F, it has malfunctioned but the outside world does not know about it. Thus, from an external point of view the module is said to be available when it is either in state W or in state F. In reality, the module should be called available only when it is in state W. Thus, we have the real availability $A_r = P_W$ and the apparent availability $A = P + P$. The purpose of the on-line detector is to keep A_a and A_r as close to each other as possible.

Note that the real availability

$$A_r = \frac{1}{1 + \frac{\lambda}{\mu} + \frac{\lambda}{\delta} + \frac{\alpha}{\beta}} = \frac{1}{1 + \lambda(\frac{1}{\mu} + \frac{1}{\delta}) + \alpha/\beta}$$

$$= \frac{1}{1 + \frac{\text{MTTR} + \text{MTDF}}{\text{MTBF}} + \frac{\alpha}{\beta}}$$

Now since α is usually much smaller than λ, we may let

$$A_r \simeq \frac{\text{MTBF}}{\text{MTBF} + (\text{MTTR} + \text{MTDF})} \tag{4}$$

Comparing expression (4) with expression (3), we conclude that MTTR + MTDF behave like an "effective" repair time. The use of a more powerful detector, i.e., a smaller value of MTDF, implies a reduction in the effective repair time, which in turn implies an increase in real availability. In fact, for fixed values of MTBF and MTTR, largest real availability results when we employ a detector with zero detection latency.

Next consider the probability of being in the undesirable state F

$$P_F = \frac{\lambda/\delta}{1 + \lambda/\mu + \lambda/\delta + \alpha/\beta} \tag{5}$$

$$\simeq \lambda/\delta\ (1 - \lambda/\mu - \lambda/\delta - \alpha/\beta)$$

$$\simeq \frac{\lambda}{\delta} (1 - \lambda/\mu - \frac{\alpha}{\beta}) - \frac{\lambda^2}{\delta^2}$$

$$\simeq \frac{\lambda}{\delta} (1 - \lambda/\mu - \frac{\alpha}{\beta})$$

thus employing a more powerful detector (i.e., increasing the value of δ) reduces P_F and hence, bringing the real availability A_r and the apparent availability A_a closer together. In fact, a detector with an infinite detection rate (or zero detection latency) implies that $P_F = 0$ and $A_a = A_r$ In this case, there is no need to distinguish between the concepts of real and apparent availabilities.

Finally, consider the apparent availability

$$\begin{aligned} A_a &= P_W + P_F \\ &= \frac{1 + \lambda/\delta}{1 + \frac{\lambda}{\mu} + \frac{\lambda}{\delta} + \frac{\alpha}{\beta}} \\ &= 1 - \frac{\lambda/\mu + \alpha/\beta}{1 + \lambda/\mu + \lambda/\delta + \alpha/\beta} \\ &= 1 - (\frac{\lambda}{\mu} + \frac{\alpha}{\beta})\ A_r \end{aligned} \tag{6}$$

Now, since A_r increases with an increase in δ (i.e., a more powerful detector), we conclude that the apparent availability reduces with an increase δ. Thus, A_a and A_r approach each other as δ increases and $A_a = A_r$ in the limit $\delta\to\infty$. Further analysis suggests that the rate of decrease in A_a is very slow since

$$\begin{aligned} A_a &\simeq 1 - (\frac{\lambda}{\mu} + \frac{\alpha}{\beta})\ (1 - \lambda/\mu - \lambda/\delta - \alpha/\beta) \\ &\simeq 1 - \frac{\lambda}{\mu} - \frac{\alpha}{\beta} \end{aligned} \tag{7}$$

Thus, as a first order approximation, the apparent availability remains constant independent of the mean detection latency.

To fix our ideas, let $\lambda = 10^{-5}$/hr, $\mu = 2$/hr, $\alpha = 10$ /hr, and $\beta = 4$/hr. In Figure 5, we have plotted the apparent availability A_a and the real availability A_r as functions of MTDF.

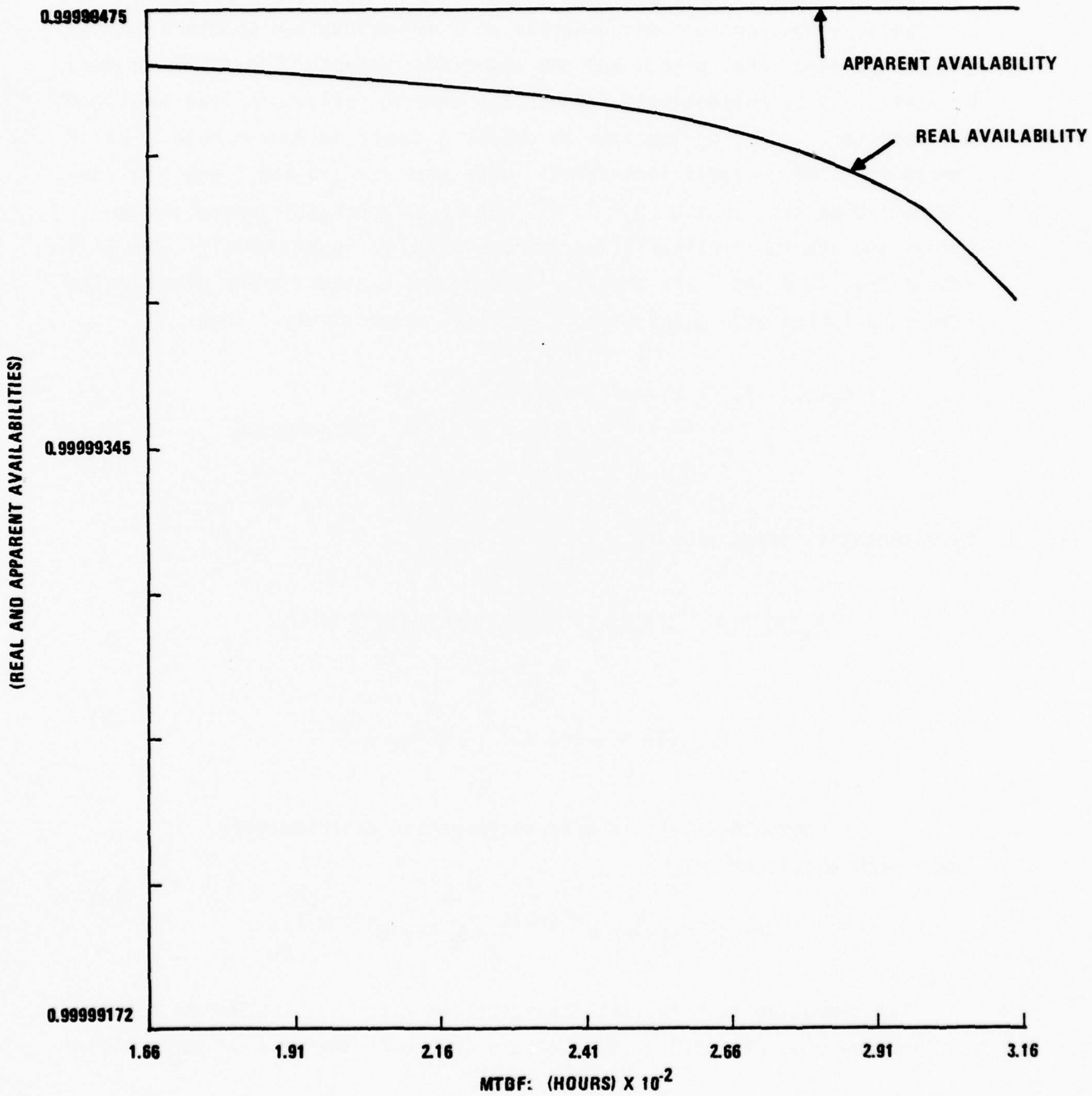


Figure 5. Apparent Availabilities vs MTDF

Analysis of a Non-Maintained Module

We will now consider the analysis of a non-maintained module M consisting of the functional unit U and the associated detector (or fault monitor) D. Let U and C, respectively, denote the time to failure of the unit and the detector. Let D be the time to detect a fault in the module. Let T denote the time to fault indication. Note that U, C, D and T are all random variables and T = min (U + D, C). Let $R_a(t)$ and $R_r(t)$ denote the apparent and the real reliabilities of the module, respectively. We will assume that U, D and C are mutually independent exponentially distributed random variables with means $1/\lambda$, $1/\delta$ and $1/\alpha$, respectively. Then

$$\begin{aligned} R_a(t) &= P(T > t) = P(U + D > t,\ C > t) \\ &= P(U + D > t)\ P\ (c > t) \quad \text{by independence} \\ &= R_U + {}_D(t)\ R_C(t) \end{aligned} \tag{8}$$

By exponential assumption,

$$R_c(t) = e^{-\alpha t},\ R_D(t) = e^{-\delta t},\ \text{and}\ R_U(t) = e^{-\lambda t}.$$

Therefore,

$$R_{U+D}(t) = \frac{\delta}{\delta - \lambda} e^{-\lambda t} - \frac{\lambda}{\delta - \lambda} e^{-\delta t} \tag{9}$$

(note that this is a hypoexponential distribution).

Then, from above, we get

$$R_a(t) = \frac{\delta}{\delta - \lambda} e^{-(\lambda+\alpha)t} - \frac{\lambda}{\delta - \lambda.} e^{-(\delta+\alpha)t} \tag{10}$$

For computing real reliability $R_r(t)$, we note that the module ceases to function properly when a fault occurs in either the unit or the detector.

Thus

$$\begin{aligned} R_r(t) &= P(U > T,\ c > t) \\ &= P(U > t)\ P(c > t) \quad \text{by independence} \\ &= R_u(t)\ R_c(t) \\ &= e \end{aligned} \tag{11}$$

We note that in the absence of the detector, the real reliability is e-λt; therefore, employing a detector actually reduces the real reliability.

Without a detector, $\alpha = 0$ and δ is near zero; therefore, the apparent reliability will be very high. Thus, the apparent reliability is also reduced by employing a detector. The purpose of an on-line detector is to close the gap between the values of the real and the apparent reliabilities.

We can also compute the real and apparent MTTF ($MTTF_r$ and $MTFF_a$):

$$\begin{aligned} MTTF_a &= \frac{\delta}{(\delta-\lambda)\quad(\lambda+\alpha)} - \frac{\lambda}{(\delta-\lambda)\quad(\delta+\alpha)} \\ &= \frac{\delta + \lambda + \alpha}{(\lambda+\alpha)\quad(\delta+\alpha)} \end{aligned} \tag{12}$$

and

$$MTTF_r = \frac{1}{\lambda + \alpha} \tag{13}$$

We define the detector effectiveness to be the ratio $MTTF_r/MTTF_a$. The detector effectiveness is plotted in Figure 6 as a function of the detection rate δ.

2.2.1.3 Design

We now present a design model for a non-maintained module. We are asked to choose the characteristics of an on-line detector that will minimize the total cost. The two cost components that enter into our model are the cost of the detector and the cost (or penalty) for the time system

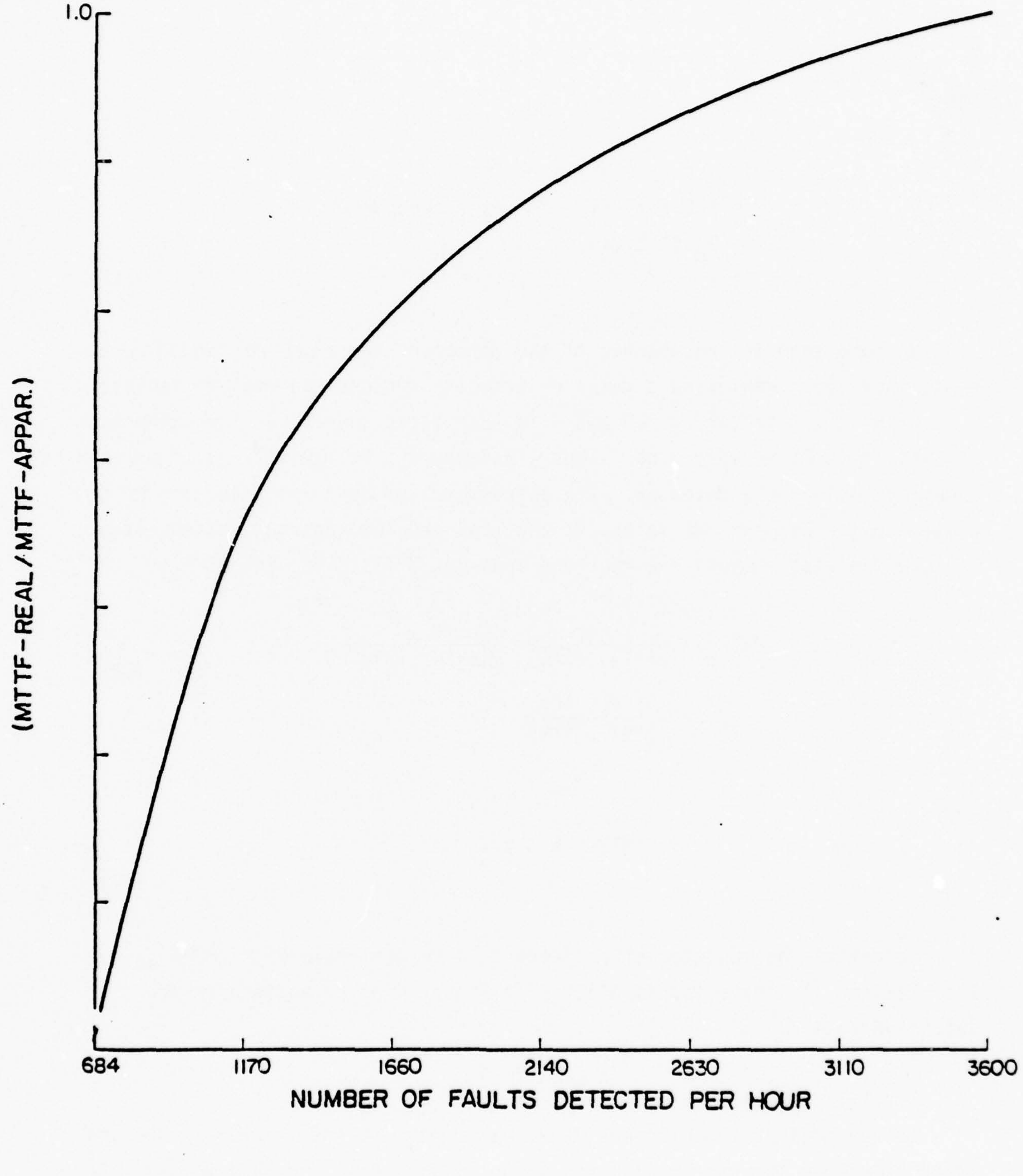


Figure 6. Detector Effectiveness

spends in the undesirable state. For the sake of simplicity, we will assume that the detector has zero failure rate, i.e., $\alpha = 0$.

The percentage of time spent by the module in the undesirable state is easily computed to be λ/δ. Let the per unit time penalty of being in this state be given by K_F. Then, the penalty is given by $K_F \lambda/\delta$. To characterize the cost of the detector, we assume that the unit U is an arithmetic unit and the detector D is a modulo-m checker (see Figure 7). The problem, then, is to determine the optimum value of m.

The cost of a modulo-m checker may be approximated by Co log m. Then the total cost

$$C = K_F\lambda/\delta + Co \log m \tag{14}$$

Note that λ, K_F and Co are assumed to be fixed parameters, but δ is expected to be a function of m.

A reasonable functional relationship is

$$\delta = \delta o \; m^a \tag{15}$$

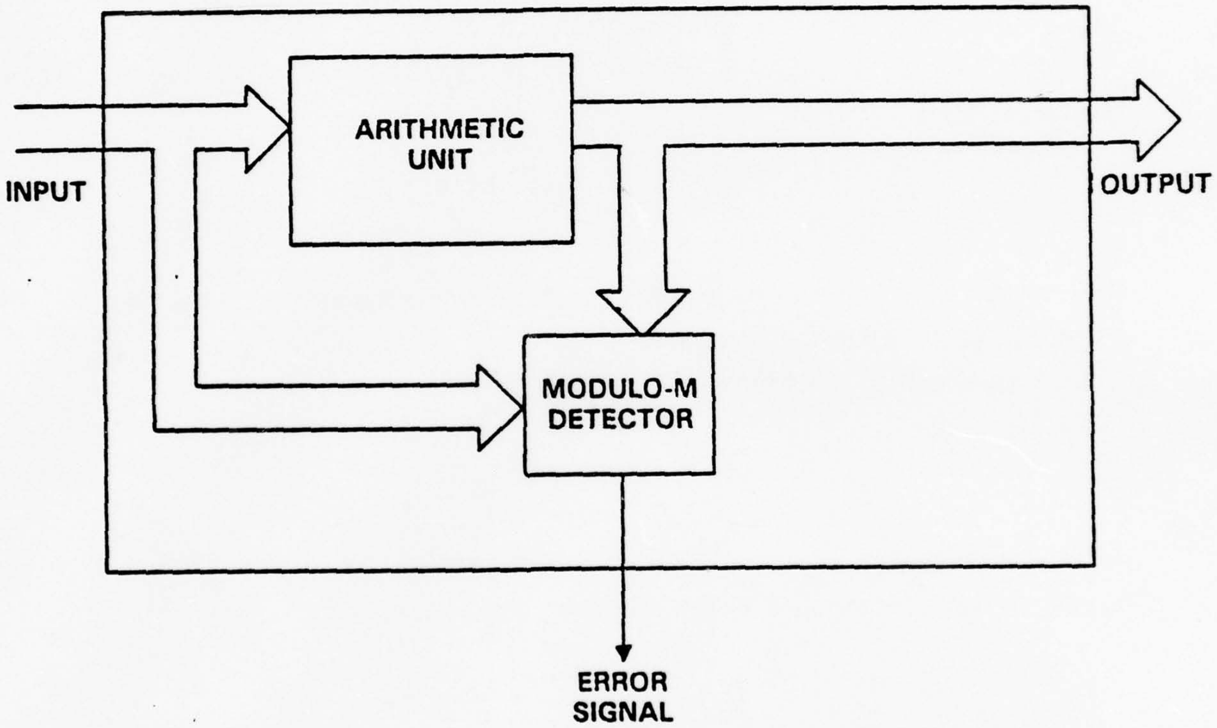
After substituting (15) in (14), we can determine the optimum value of m by taking $\frac{dc}{dm}$ and setting it equal to zero:

$$\frac{dc}{dm} = - \frac{aK_F\lambda}{\delta o \; m^{a+1}} + \frac{Co}{m} = o$$

$$\text{or} \quad m_{opt} = \sqrt{\frac{K_F}{C_o} \cdot \frac{\lambda a}{\delta_o}} \tag{16}$$

This shows that the larger the value of K_F relative to Co, the larger should be the value of m. In other words, if the penalty of being in the undesirable state is large, we should choose a more powerful detector.

It is hard to parameterize such a model. We obtained data from a paper by J. Clary [5] and fitted the data to obtain the values of δo and a.



PROBLEM IS TO DETERMINE THE CHECK MODULUS M THAT WILL MINIMIZE THE COST.

Figure 7. Design of a Non-Maintained Module

Using these values and $\lambda = 10^{-5}$/hr, we can determine the optimal value of m as a function of the relative cost K_F/Co from equation (16). This function is plotted in Figure 8.

2.2.1.4 Work Planned for the Future

First, we plan to consolidate all the above models of systems analysis. In addition to the four-state model presented in Section II, we will also include models with multiple fault types and models with multiple on-line detectors. We will also include models of non-repairable systems.

The next task to be undertaken is the development of system design models for a repairable module. The problem is to choose the rates λ, μ, δ and α so as to maximize real availability subject to a cost constraint. The cost of the module consists in the cost of the unit, the cost of the detector, the cost of the repair facility, and the cost associated with the module state F. Recall that in state F, the module has failed but the failure is not yet detected. Such a state of the module may be very harmful and there may be a heavy penalty associated with it. The main problem in such a model is how to characterize various cost components as functions of the decision variables. We plan to spend a good deal of time on this phase of the research and we anticipate very interesting and useful results.

Many extensions to the models for analysis and the models for design are evident to us. We may include various types of static, standby or hybrid-redundant schemes in our models. We may also remove the all pervasive assumption of independence and consider a system consisting of associated components [3]. Parameterization and actual use of these models to analyze real systems is also an extensive and interesting project.

The work proposed in this section may take two or more years for its completion. The help of at least one and perhaps two graduate student assistants is desirable for this massive effort. We may note that to train prospective students to work along these lines, a course developed by this author in the Department of Computer Science at Duke University is very helpful. This course is titled, "Probability Theory and Applications to

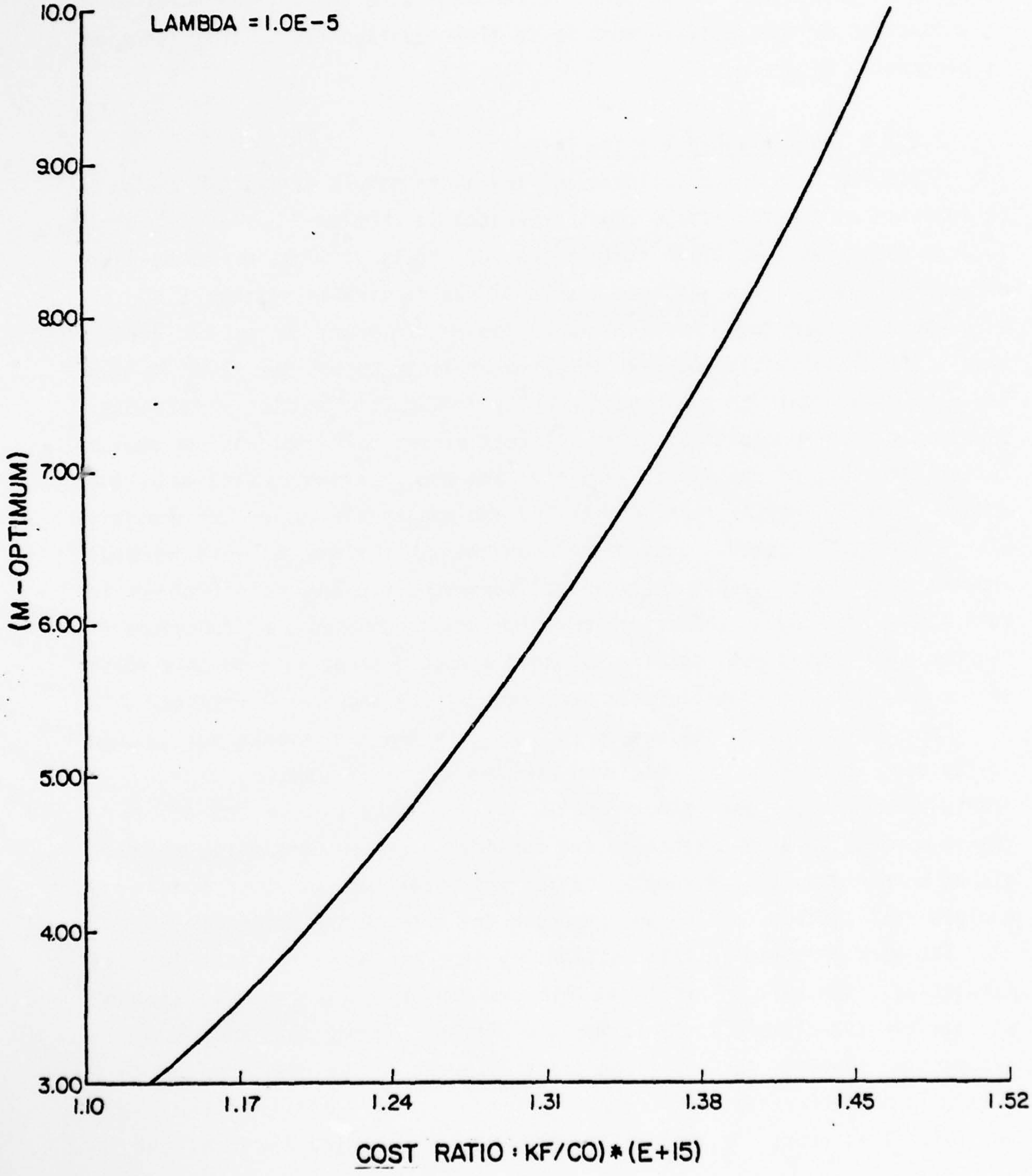


Figure 8. Optimum Check Modulus

Computer Science and Electrical Engineering." A good part of the course is devoted to stochastic models of system reliability and availability.

LIST OF REFERENCES

1. Clary, J. B., Gault, J. W., Weikel, S. J., Whisnant, R. A., Alberts, R. D., "A Study of a Standard BIT Circuit," Final Report Contract N00163-76-C-0231, Research Triangle Institute, Research Triangle Park, N.C., 1977.

2. Clary, J. B., Gault, J. W., Marinos, P. N., Trivedi, K. S., Whisnant, R. A., and Alberts, R. D., "Basic Research in Support of Concurent Fault Monitoring in Modular Digital Systems," Proposal, Contract No. N00039-77-PR-7J-133, Research Triangle Institute, Research Triangle Park, N.C., 1977.

3. Barlow, R. E., and Proschan, F., Statistical Theory of Reliability and Life Testing: Probability Models, Holt, Rinehart and Winston, New York, 1975.

4. Mathur, F. P., and Avizienis, A. A., "Reliability Analysis and Architecture of a Hybrid-Redundant Digital System: Generalized Triple Modular Redundancy with Self-Repair," in 1970 Spring Joint Computer Conf., AFIPS Conf. Proc., Vol. 36. Montvale, N.J.: AFIPS press, 1970, pp. 375-383.

5. Clary, J. B., "Effectivensss Measures for Built-In-Test Performance Evaluation," Technical Report, Research Triangle Institute, Research Triangle Park, N.C., 1977.

2.2.2 BIT FACILITY IDENTIFICATION AND EVALUATION

By

Dr. Peter N. Marinos
Professor of Electrical Engineering
and Computer Science
Duke University, Durham, N.C.

ABSTRACT

This part of the report represents two major undertakings. First, it describes a unique, programmable cellular structure capable of realizing any arbitrary sequential machine; and second, it presents the design and detail description of a high-level digital computer simulator with fault injection facilities.

The motivation for developing this programmable cellular structure was the importance of modular design in achieving improvements in digital system reliability and availability while retaining system flexibility. The proposed basic cell is so configured that it makes the design of Built-In-Test (BIT) facilities a natural extension of the overall system design process. The approach taken here in arriving at a testable cellular structure is referred to as "hardware encoding" to distinguish it from more traditional information encoding schemes. The "hardware encoding" of the cellular structure relies on the same basic cell used to configure the rest of the cellular structure and may be thought of as the hardware analog of well-known information encoding procedures.

The high-level digital computer simulator with fault injection facilities, which was developed as a Master's thesis in Computer Science at Duke University under the supervision of Professor P. N. Marinos, represents a very useful tool in evaluating the effectiveness of various BIT facilities. The unique feature of this simulator is its ability to combine the functional flexibility of a simulated hardware organization with the ability to process a typical work load on the simulated machine subject to a specified fault environment.

2.2.2.1 Background

The reliability and availability of digital systems can be increased by the use of built-in-test (BIT) facilities capable of detecting all system faults of a specified class. Desirable properties to be possessed by such BIT facilities are:

1. Self-checkability,
2. General applicability,
3. Fault resolvability to a specified modular level,
4. Suitability for use with current technologies (i.e., LSI),
5. Fault management and fault reporting capability for the purpose of effecting system recovery (i.e., system repair and/or system reconfiguration), and
6. Passive (i.e., non-interfering), continuous system monitoring capability, at least until a fault has been detected, at which time the BIT facility may become active and participate in system repairs.

The objectives of our task are specifically:

1. The development of BIT facilities for use at the system and subsystem level with special emphasis on the implementation and distribution of such facilities throughout the entire system, and
2. The evaluation of the effectiveness of various BIT facilities in terms of added cost and complexity required for their implementation, as well as in terms of the level of system protection they provide.

Work performed during the period September 1, 1977 to May 15, 1978 has considered both objectives outlined above, and it is presented in the sequel as Part-A and Part-B. Part-A describes a programmable cellular structure with "hardware encoded" BIT facilities while Part-B presents a high-level digital computer simulator with fault injection facilities.

2.2.2.2 Programmable Cellular Structures With Hardware Encoded BIT Facilities

Introduction

The advantages of cellular arrays, which result from the regularity of their iterative structure, have been sufficiently documented [1-7]. Memory manufacturers have fully exploited many unique features of cellular arrays, and the presently realizable memories with highly improved device-densities production yields, size, cost, speed, noise, and reliability constitute excellent testimony of their success.

Since the primary motivation for integrated circuits (IC's) is the reduction of interconnections and packages, which in turn translates into a number of improvements in terms of cost, power, speed, size, noise, and reliability, it is quite natural for IC manufacturers to look into cellular structures as the next natural step towards bringing about further improvements in system integration, system reliability, and system testability and maintainability. This implies incorporation of cellular logic design notions into the design of what has been traditionally known as random logic subsystems such as control units and arithmetic and logic units of digital systems.

There are three main reasons why random logic subsystems have remained largely non-cellular in form, namely: lack of design standardization; inadequate testing procedures for fault detection and fault diagnosis; and poor maintenance schemes in terms of self-reconfiguration in the event of failure, as well as in terms of preventive maintenance. These are major problem areas and are currently attracting a great deal of attention in sev eral industrial and university laboratories. This interest is easily justified by the fact that cost, device densities and yields of mass-produced cellular structures using mature technologies have greatly improved over the years, and we find ourselves now in the comfortable position of being able to build very economically into such cellular structures, redundant functional capability, which one could use to improve system integration, system testability and maintainability, and, in general, system reliability and availability.

In recent publications, Manning [6] describes certain procedures for automatic testing, configuration, and repair of cellular arrays; and Page and Marinos [7] propose a programmable array for use in designing synchronous sequential machines, and demonstrate ways for actually embedding arbitrary finite-state machines in such arrays. What is proposed next is a natural extension of these two independent research efforts, and it is based on the strong belief that industry will recognize the many advantages of us ing cellular structures throughout a digital system once the issues of array standardization (both functional as well as structural), testability and maintainability have been resolved in a practical and cost-effective manner

Cellular Arrays Utilizing K-Out-Of-M State Assignments

The cellular array envisioned in this study is comprised of programmable logic cells capable of supporting all the combinational logic needs of a system, and of memory cells either in a physically separate cellular array or as part of the overall programmable cellular structure. The separation of the combinational logic from the memory unit arises very naturally in sequential machine design and it is additionally justified by the fact that testing procedures for memories are distinctly different from those used for testing the non-memory portion of a system. For these reasons, and the fact that memories in cellular form will always be available, independently of how one decides to implement the random logic of control units and arithmetic-logic units, we chose to maintain this separation and not to distribute memory among the cells of the programmable cellular structure.

The proposed cellular array represents a basic logic structure one can successfully utilize to realize, in a programmatic way, any arbitrary sequential machine. The design approach made possible by the proposed array is highly modular and offers many opportunities for achieving improvements in digital system reliability and availability while retaining system flexibility. The incorporation of Built-In-Test (BIT) facilities in systems configured from the proposed cellular arrays is a natural extension of the overall system design process requiring no special hardware considerations.

The main objective of this study is to develop and distribute the BIT facilities over the cellular structure of systems based on the proposed cellular array and to evaluate the effectiveness of such facilities in terms of added system cost, system reliability, system availability, and system maintainability. The design of the basic cell utilized in the cellular array has been motivated by the requirements and properties of the BIT facilities which were outlined earlier.

Characterization of Synchronous Sequential Machines: A finite-state synchronous sequential machine is described by the algebraic structure

$$M = < X, Z, Q, \delta_j, \omega > \text{, where}$$

X = a finite set of input symbols $(x_1, x_2, \ldots, x_n)$ such that $x_i \epsilon (0,1)$, $i = 1, 2, \ldots, n$;

Z = a finite set of output symbols $(z_1, z_2, \ldots, z_p)$ such that $z_j \epsilon (0,1)$, $j = 1, 2, \ldots, p$;

Q = a finite set of states $(q_1, q_2, \ldots, q_t)$ defined by the state variables $(y_1, y_2, \ldots, y_m)$ such that $m \geq \log t$;

$\delta : X \times Q \xrightarrow{\text{into}} Q$ is the next-state function

$$\left.\begin{array}{ll} \omega : & X \times Q \xrightarrow{\text{onto}} Z \\ \text{or} & Q \xrightarrow{\text{onto}} Z \end{array}\right\} \quad \text{is the output function}$$

The general form of the excitation and output functions of a sequential machine may be written as follows:

$$F_r(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m) = \qquad (1)$$

$$\bigcup_{i=1}^{t} f_{i,r} (x_1, x_2, \ldots, x_n) \cdot q_i$$

where

$$r = 1, 2, \ldots, m, \ldots, m + p$$

$q_i = \overset{*}{y}_1\, \overset{*}{y}_2 \ldots \overset{*}{y}_m$, with $\overset{*}{y}_j$ denoting the state variable y_j either in its complemented or uncomplemented form (i.e., q_i denotes a min-term of the state variables y_j, $j = 1, 2, \ldots, m$.)

For reasons outlined elsewhere [7,8], state assignments based on monotone, k-out-of-m codes will be utilized. Among the many advantages offered by these codes, the one of interest in this case is their utility in error detection, and in designing fail-safe sequential machines [9,10]. In view of such a state assignment, one may rewrite equation (1) in the form,

$$F_r = \bigcup_{i=1}^{t} f_{i,r}\,(x_1, x_2, \ldots, x_n) \cdot G_i(m,k) \qquad (2)$$

where

$$G_i(m,k) = y_{i_1} y_{i_2} \ldots y_{i_k}$$

and $y_{i_j}\ \varepsilon(y_1, y_2, \ldots, y_m)$, $j = 1, 2, \ldots, k$.

Equation (2) suggests a linear array, each cell of which is algebraically described by the function

$$A_{i,r} = \S_{i,r} = A_{i-1,r} + f_{i,r}\,(x_1, x_2, \ldots, x_n)\, G_i(m,k) \qquad (3)$$

Figure 1 shows the structure of such a cell with an n-input programmable universal logic module (PULM-n) implementing the function $f_{i,r}\,(x_1, x_2, \ldots, x_n)$. The PULM-n unit is programmed via the associated programming register. The linear cellular array shown in Figure 2 implements the function F_r given

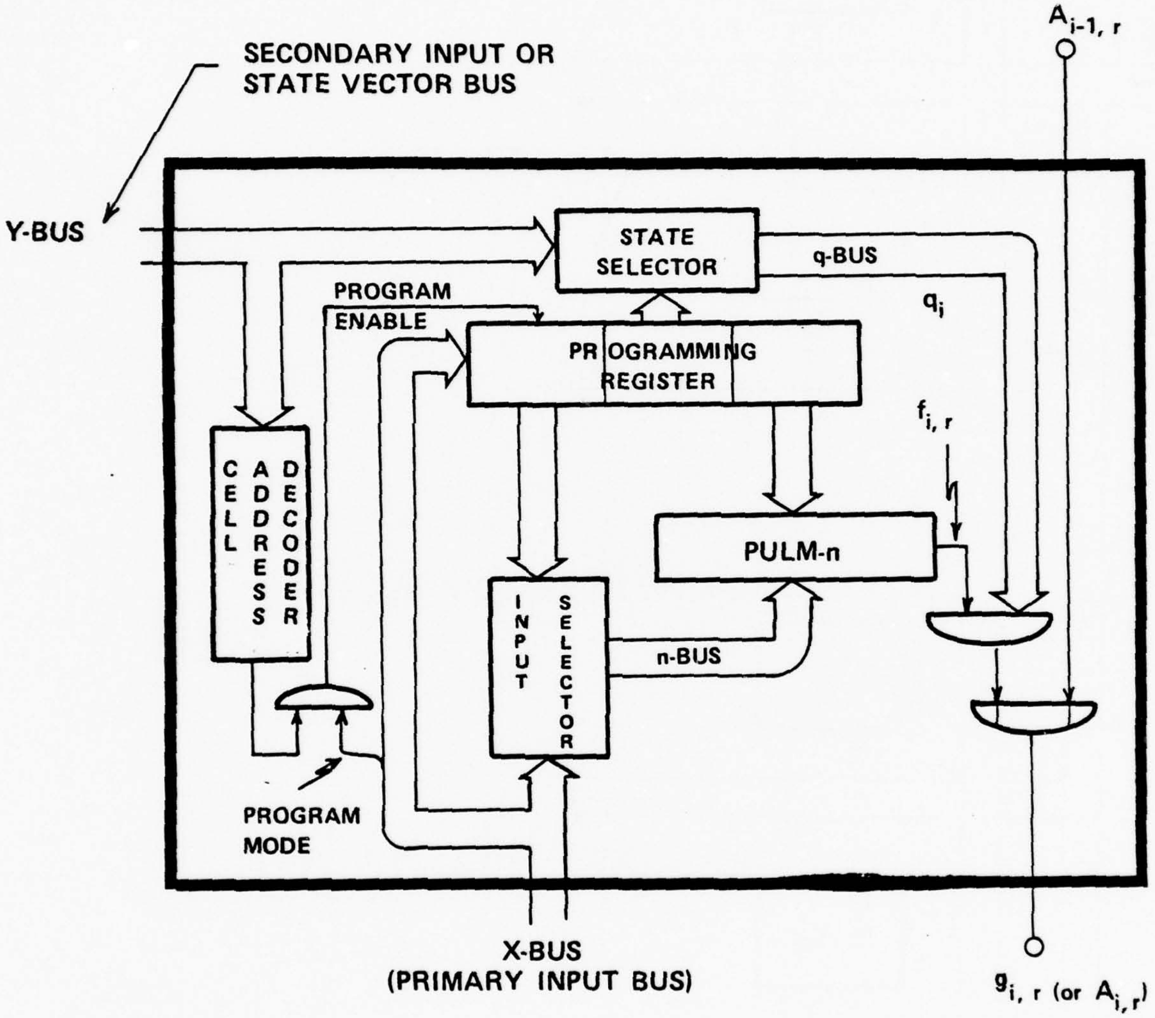


Figure 1. Basic Cell

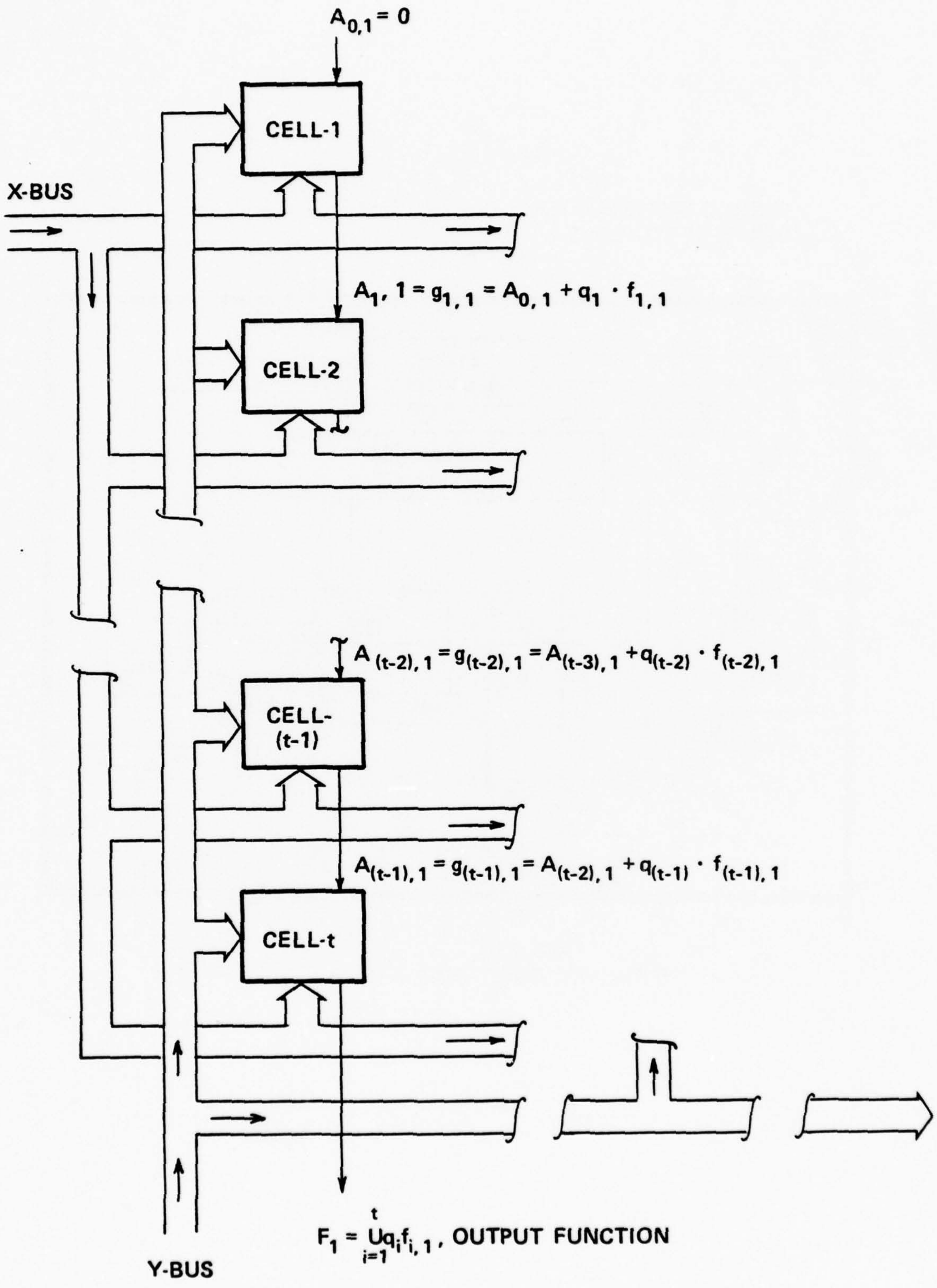

A0,1 = 0
CELL-1
X-BUS
A1, 1 = g1, 1 = A0, 1 + q1 · f1, 1
CELL-2
A (t-2), 1 = g(t-2), 1 = A(t-3), 1 + q(t-2) · f(t-2), 1
CELL-(t-1)
A(t-1), 1 = g(t-1), 1 = A(t-2), 1 + q(t-1) · f(t-1), 1
CELL-t
F1 = U(i=1..t) qi fi, 1 , OUTPUT FUNCTION
Y-BUS

Figure 2. A Cellular Cascade

by expression (2), and the two-dimensional cellular structure given in Figure 3 illustrates a cellular array capable of implementing any required, finite number of excitation and output functions F_r necessary for the imple-ementation of a finite state, synchronous sequential machine. One of the structural advantages of the array shown in Figure 3 is its ability to support finite state, synchronous sequential machines of arbitrary "state-space" cardinality without requiring any structural changes in the basic cell of the array.

Description of Basic Cell: The major module of the basic cell shown in Figure 1 is the so-called programmable universal logic module (PULM-n) capable of realizing any switching function of n binary variables. The programmability of this module is made possible through an appropriate field in the programming register while the n binary variables used by the module to form the desired n-variable switching function are provided via an input selector and accessed through the n-BUS under the control of the programming register. Finally, a third field in the programming register is used to select the state, q_i, associated with the cell in question.

The Input-Selector and State-Selector units receive primary input and secondary input (or state) information via the X-BUS and Y-BUS, respectively. These two busses may be arbitrarily large in size while the q-BUS and n-BUS, referred to earlier, are, for practical considerations, of significantly smaller size. In the case of the n-BUS, this is motivated by the growth in size of the PULM-n unit as n becomes larger; with respect to the q-BUS, one is interested in satisfying the state-space of a machine with the minimum number of state variables. For K-out-of-m codes used here, the choice of K, which denotes the size of the q-BUS, must be such that it produces the largest possible number of states from the m state variables brought in via the Y-BUS. Thus, the usual choice for K is such that the expression $\frac{m!}{k!\cdot(k-m)!}$ is maximized. It should be noted that with the X-BUS there is an extra line known as the "program mode" line used for controlling the programming register, and its specific function will be revealed shortly.

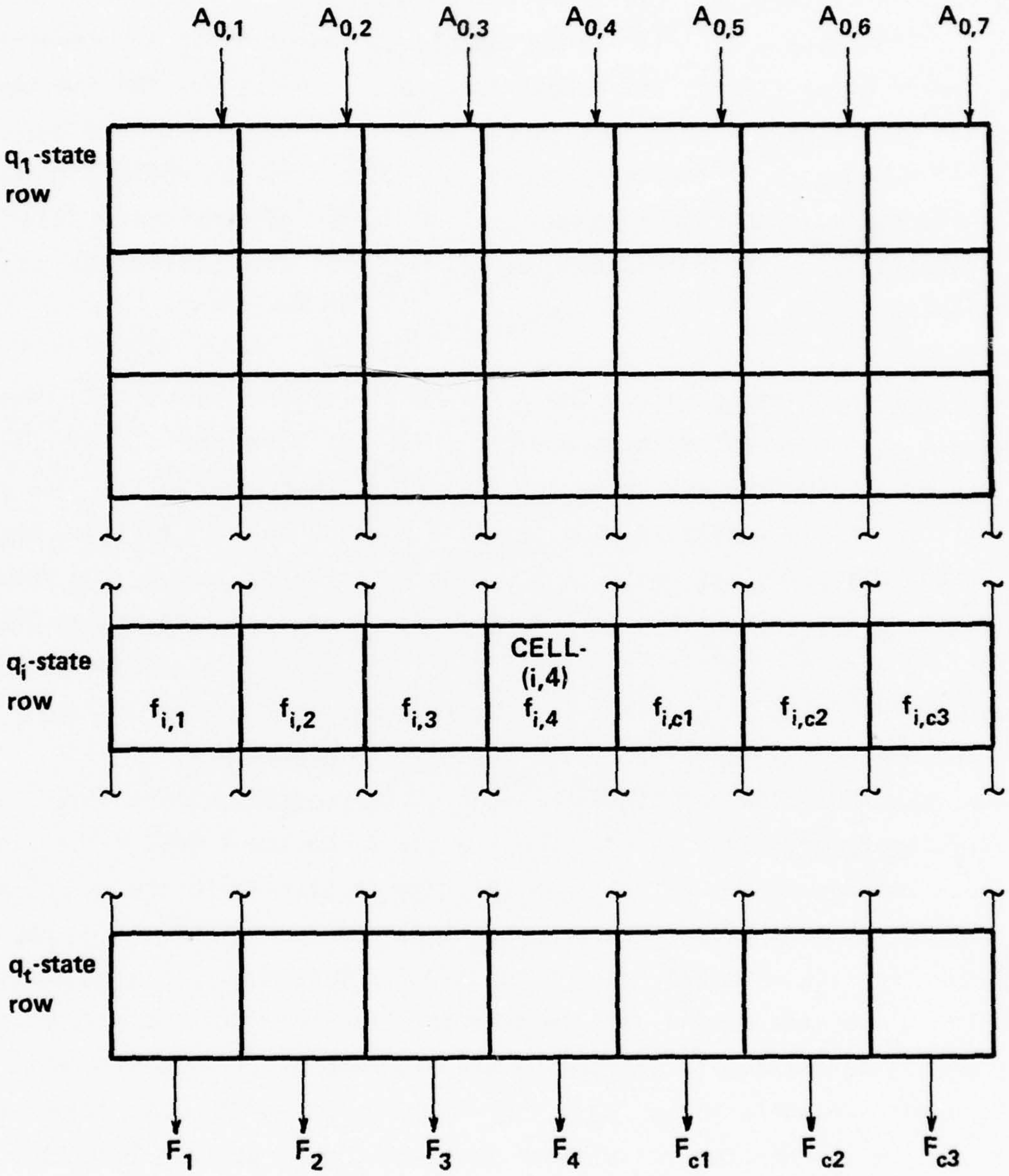


Figure 3. Encoded Cellular Array

A unique feature of the proposed basic cell is the dual function served by the X- and Y-BUSSES. In addition to their main function of importing primary and state information to the cell, they also serve two additional functions: The X-Buss is used to program the programming register, provided the cell in question has been properly addressed via the Y-BUS to generate the required "program enable" control signal, which is essential in reprogramming the PULM-n module. Thus, the second function served by the Y-BUS is to address uniquely the cell of interest by making use of a unique "cell address decoder." The dual functions served by the X- and Y-BUSSES constitute a very important design consideration since they help maintain a low pin-count for the cell. In the "program mode" the cell output is disregarded.

Another important feature of the cell in Figure 1 is the fact that all the incoming information is reduced or "fully decoded" to a single-line output which is very important from the standpoint of pin-count.

Description of Cellular Array: An aggregate of basic cells arranged as shown in Figure 2 forms a cellular array. Each cell has full access to the X- and Y-BUS, and function realization is effected along a column. The upper boundaries of the array are set to zero, and the realized function along each column is output as F_i and represents either an excitation or output function.

Each cell in a column is programmed to account for a specific term of the function which is implemented in a sum-of-products form. For functions which are independent of state information, the "state selector" in each cell is capable of providing the Boolean constant 1 under control from the appropriate field of the programming register, thus facilitating the generation of product terms not requiring state information.

Using the algorithmic procedure developed by Page and Marinos [7], one may now embed any arbitrary, synchronous sequential machine in the cellular array of Figure 2. The cellular array described in this study utilizes a basic cell which is functionally more complex than the one used by Page and Marinos [7], and, as a result, it leads to more flexible cellular structures. It should be noted that the proposed cellular array permits localized selection of both state variables and primary input variables and facilitates its own programmability via the primary and secondary input busses X and Y. Each cell in the array is individually addressed, and the array is assumed to obtain its state information from a separate memory array.

Cellular Array With Hardware Encoded BIT Facilities

The term "hardware encoding" is used to mean "the process of mapping in a systematic way any of the well-known information encoding schemes onto a cellular array structure."

The manner in which such a mapping is carried out is greatly simplified by assuming that all the cells of a row are programmed to receive the same state code. This is only a topological and not a functional restriction im posed on the machine which is embedded in the cellular array and it is made for the sake of analytical convenience. Thus, the mapping of a sequential machine into a cellular array, using the algorithm by Page and Marinos [7], results in a cellular machine structure in which each row is uniquely associated with a machine state, except in the case of state splitting to be discussed later.

Figure 3 illustrates a cellular array in which the i^{th} row is associated with machine state q_i. The busses and other details of the cellular array are omitted for the sake of clarity. The four leftmost columns of the array are used to implement machine functions (i.e., excitation and output functions) while the remaining three (i.e., F_{c1}, F_{c2} and F_{c3}) denote the built-in check functions employed, in this case, to encode the information carrying functions F_1, F_2, F_3 and F_4 according to the well-known single-error correcting Hamming code. These checking functions are implemented in the last three columns of the array in a manner similar to that used to form any other machine function. The resulting array is one with "hardware encoded" BIT facilities that make continuous and non-interfering monitoring of the encoded output (i.e., $F_1 \; F_2 \; F_3 \; F_4 \; F_{c1} \; F_{c2} \; F_{c3}$), using standard error checking schemes, possible.

Although the illustration in Figure 3 uses a single-error-correcting Hamming code, other encoding schemes may be just as easily employed by utilizing the appropriate relations which must hold between the "information cell" function $f_{i,r}$'s and the "checking cell" functions $f_{i,cj}$'s.

For the single-error-correcting Hamming code used in Figure 3, the well-known relationships are given by the equations

$$\begin{aligned} f_{i,c1} &= f_{i,1} \oplus f_{i,2} \oplus f_{i,4} \\ f_{i,c2} &= f_{i,1} \oplus f_{i,3} \oplus f_{i,4} \\ f_{i,c3} &= f_{i,2} \oplus f_{i,3} \oplus f_{i,4} \end{aligned} \qquad (4)$$

assuming, of course, even parity. The above equations prescribe the respective "checking cell" functions that must be programmed so that when the machine is in state q_r, the r^{th} row of the array is "hardware encoded" in accordance with the coding scheme of interest.

In the case of a combinational (i.e., memoryless) network, the checking functions are realized in a manner similar to the sequential network case by assuming the network as being a one-state machine. Whenever the functional complexity of a machine exceeds the resources of a row in a cellular array, then more than one row may be associated with a single state resulting in what we have previously referred to as state splitting.

The approach outlined above for realizing BIT facilities is compatible with LSI technologies, and it makes full utilization of well-known information encoding schemes used for error detection and diagnosis in digital systems. It is worth noting that "hardware encoded" BIT facilities, as proposed here, do not require specially designed hardware, and they are incorporated into the overall system in a way that makes them a natural extension of the non-encoded machine structure.

There are many unresolved questions concerning optimal machine layout and mapping onto a cellular array; similarly, there are problems with respect to encoding schemes suitable for various machine layouts. These and other issues, such as distribution of spare cells over a cellular array and maintenance policies, which impact the cost-effectiveness and reliability of such arrays, are the object of our continuing research efforts in this area.

REFERENCES

1. R. C. Minnick, "Cutpoint Cellular Logic," IEEE Trans. on Electronic Computers, vol. EC-13, pp. 685-698, Dec. 1964.

2. R. C. Minnick, "A Survey of Microcellular Research," J. Assoc. Comput. Mach., vol. 14, pp. 203-241, April 1967.

3. E. F. Codd, "Cellular Automata." New York and London: Academic Press, 1968.

4. W. H. Kautz, "Testing for Faults in Combinational Cellular Logic Arrays," Proc. 8th Ann. Symp. Switching and Automata Theory, Oct. 1967, pp. 161-174.

5. S. S. Yau and M. Orsic, "Fault Diagnosis and Repair of Cutpoint Cellular Arrays," IEEE Trans. on Computers, vol. C-19, no. 3, pp. 259-261.

6. F. B. Manning, "An Approach to Highly Integrated, Computer-Maintained Cellular Arrays, "IEEE Trans. on Computers, vol. C-26, no. 6, pp. 536-552, June, 1977.

7. E. W. Page and P. N. Marinos, "Programmable Array Realization of Synchronous Sequential Machines, "IEEE Trans. on Computers, vol. C-26, No. 8, pp. 811-818, August, 1977.

8. S. Mago, "Monotone Functions in Sequential Circuits," IEEE Trans. on Computers, vol. C-22, pp. 928-933, Oct., 1973.

9. T. Takaoka and T. Ibaraki, "N-Fail-Safe Sequential Machines," IEEE Trans. on Computers vol. C-21, pp. 1189-1196, Nov. 1972.

10. Y. Tohma, Y. Ohyama, and R. Sakai, "Realization of Fail-Safe Sequential Machines by Using K-out-of-m Code," IEEE Trans. on Computers, vol. C-20, pp. 1270-1275, Nov. 1971.

2.3 MODULAR COMPUTER SYSTEMS WITH SOFTWARE UE-HANDLING AND HARDWARE BUILT-IN-TEST

D. L. Parnas, Principal Investigator
Professor of Computer Science
University of North Carolina

D. H. Bowles
Graduate Research Assistant
University of North Carolina

This work is based upon the premise that the addition of Built-In-Test (BIT) hardware and software for the handling of undesired events (UEs) may increase reliability in modular computer systems, i.e., increase the likelihood that the computer system will help us when we need its help. Currently, many software engineers espouse a "zero defects" philosophy, one that expects software to be correct. Such a philosophy is impractical; even though program verification and structured programming are useful tools, UEs will almost invariably occur. Our proposed approach to system design has three main components: *anticipation* of the occurrence of possible UEs; *detection* of occurring UEs; *response* to the UEs (correction where possible). Such an approach should be useful in the development of reliable modular computer systems.

The specific project subtask is to examine the hardware/software interface requirements in modular computer systems that support the handling of UEs by software and to provide generally useful guidelines for the specification of such interfaces. In addition to the previous work that Dr. D. L. Parnas has done on this subject, other research has been done primarily in Europe. As a result, the most comprehensive research report known to us was written in German. Our earliest efforts were directed, therefore, to making this literature available in English. (Reports of previous research concerning BIT were provided by the Research Triangle Institute.)

The most relevant sections of the research report, *Reaktion auf Unerwuenschte, Ereignisse in Hierarchisch Strukturierten Software-Systemen* (Reaction to Undesired Events in Hierarchically Structured Software Systems), by Dr. H. Wüerges, have been translated into English (refer to Section 2.3.1). Wüerges, who worked closely with Dr. D. L. Parnas, describes the methods for handling UEs, paying particular attention to hardware/software interfaces and cost factors. Of major interest are his suggestions for hardware/software interfaces (reference Chapter 12 of Wüerges' thesis). Wüerges' experimental work was based on a Siemans 4004 architecture. In his thesis, he demonstrates how hardware design errors presented effective software recovery.

The fundamental issues that are addressed in the research are: the redundancy of code and of information; distinction between functions best performed by the hardware and those best performed by the software; the specification of the hardware/software interface. For example, the redundancy of data in a system in which the frequency of system failure is high may be considerably more economical than the costs incurred by the loss of the data.

We have modified and/or expanded many of the ideas contained in Wüerges' thesis. For example, anticipation of the occurrence of possible UEs plays a major role in our philosophy of system design -- one can not handle a UE whose occurrence has not been anticipated. Wüerges provided a classification scheme for possible UEs that was useful for his purposes, but which was rather limited in scope; i.e., the classification is peculiar to the machine on which he conducted his experimental work. A more general handling of the topic required a precise specification of the possible UEs. Such a classification has been developed in order to provide a comprehensive base for the work being undertaken.

Three classes of fault sources are distinguished: the hardware, the software and external sources. Among hardware defects are defective electronic module, defective I/O transducer, defective interconnection, and crosstalk and noise. The definitions of the first three depend upon the partitioning of the system. Software faults include incomplete coding and violation of the applicability conditions. Power source, defective offline storage media, operator (including system configuring and control) and input data are external fault sources. As important as the fault sources are the ways in which faults manifest themselves. As an example, the failure of a bit in a location where a program segment is stored may manifest itself as a program error, i.e., improper or undefined operation or operand code. Possible fault manifestations are given in Table 1.

Preparation for UEs is a rather useless action without the ability to detect them. Wüerges assumed no BIT -- our research is based on a system which includes BIT hardware and software modules for the detection of UEs and for the provision of information concerning occurring UEs. Detection of a failure (by the BIT hardware module) generates a high-priority interrupt to the functional hardware and causes a transfer of control to the BIT software module. (Refer to Figure 1 for characterization of the relationships between the various hardware and software modules).

Table 1

Fault Manifestations

I. HARDWARE

A. INCORRECT MODULE OPERATION

B. INCORRECT I/O TRANSDUCER OPERATION

C. FAULTY DATA TRANSFER BETWEEN MODULES

D. FAULTY DATA TRANSFER FROM I/O TRANSDUCERS

E. UNRESPONSIVE I/O PORT

F. UNREASONABLE OR INCORRECT REGISTER* CONTENTS

G. UNREASONABLE VOLTAGE OR CURRENT LEVELS

II. SOFTWARE

A. UNREASONABLE OR INCORRECT REGISTER** CONTENTS

B. INCORRECT INPUT DATA ON FAULTY INPUT DATA TRANSFER

III. EXTERNAL

A. PATHOLOGICAL DISPLAY

B. PATHOLOGICAL OUTPUT DATA

C. SYSTEM CRASH

---

* AU REGISTERS, INCLUDING MEMORY

** SOFTWARE - ACCESSIBLE REGISTERS ACCUMULATOR, INDEX REGISTERS, STACK POINTER, ETC.

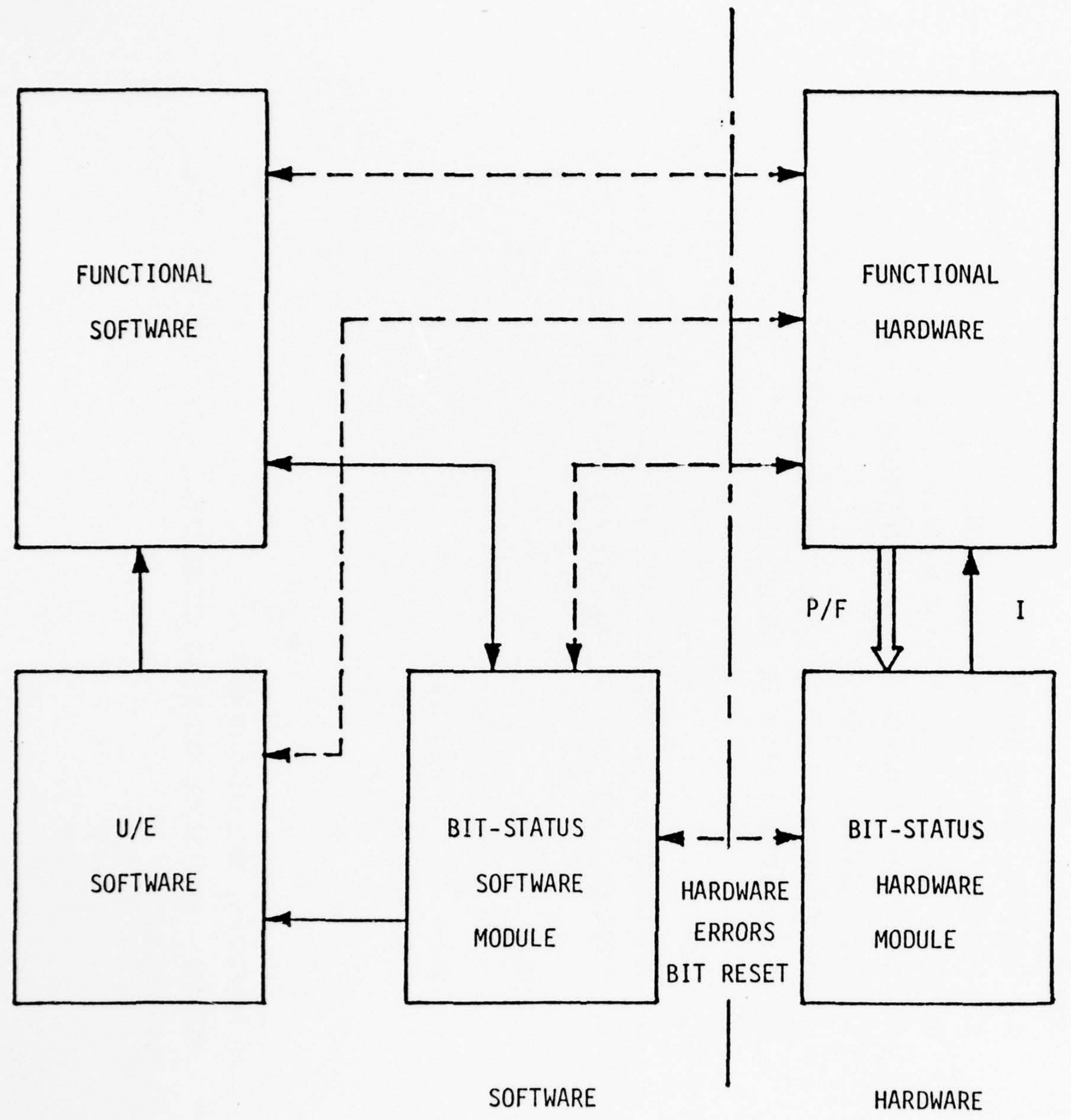


Figure 1. Programmable Computer With Undesired Event Software and Built-In-Test Hardware

Response to detected UEs is the third facet of our approach to the design of reliable modular computer systems. Hardware-detected UEs will be reported by means of traps, as described in Wüerges' thesis. Similar UEs would be handled by a single parameterized routine; the number of such routines is dependent upon the explicit classification of hardware-detected UEs, the system hardware and costs. Special care will be taken to preserve the integrity of the various modules, as described by the "information-hiding" principle of Parnas (reference CACM, December 1972). Initially, basic information will be provided about as many UEs as it would be possible for us to detect and "handle"; i.e., checks for many UEs are provided. As outlined in Wüerges' thesis, the separation of code for the normal case and for the UE case are strictly maintained; this allows easy and independent modification of either. Checks for certain UEs will be removed when experience indicates that these UEs occur only infrequently. This will reduce the additional costs incurred by the inclusion of UE-handling tools. After initial system testing is performed, the UE-handling routines can be modified to allow desired capabilities.

Part of the work is devoted to the determination of the best means of providing necessary functions, i.e., whether by the hardware or by software. The basic premise of UE-handling is that a UE is best "handled" at a level in which it is caused. If a UE is detected at a lower level (here the hardware), UE-handling is attempted at each higher level in the hierarchy. This requires reconstruction of the state of the program at each level. If no level has sufficient information to respond to the UE, termination results at the level of the ultimate user. Of major concern is the provision of additional registers and/or reserved memory locations, additional software (including the possibility of software to perform some hardware functions) and modified or additional instructions. Such decisions, of course, depend upon performance-cost trade-off. Particularly noteworthy is the addition of three instructions described by Wüerges:

1. CONTINUE - If recovery from a UE is possible, then continue execution at the instruction immediately following the action in which the UE occurred.

2. RETRY - Attempt to repeat the action in which the UE occurred, if the occurrence of the UE is not "fatal."

3. CLEAR - The user does not wish to continue the interrupted operation, in which case the effects of the operation must be removed.

These are some of the basic tools needed to allow for recovery from UEs in computer systems.

We have examined the specifications for the hardware/software interface and propose to specify the hardware/software interface for a simple modular system, following the guidelines proposed by Wüerges and expanding them as described above. The basic model consists of an Intel 8080 CPU, a read-only memory, a random-access memory and a BIT status module (refer to Figure 2). The model is particularly appropriate for our needs, since almost no error-handling capabilities are provided (only an external interrupt). In specifying the hardware/software interface, software for UE-handling may be introduced in some programs, yet programs that would have run on the system without this feature may still be run.

A simulator that implements a subset of the Intel 8080A instruction set has been provided by Bowles. A modification of this simulator, in addition to simulations of other components of the system, may be used for the purposes of system testing. Proposed changes to the Intel 8080A include the addition of: traps (refer to Wüerges' thesis); the commands CONTINUE, RETRY, CLEAR; hardware-implemented BIT. The simulation of the entire system and system testing had been scheduled tentatively for the summer.

The primary goal of this endeavor is total system reliability, i.e., the development of systems in which undesired events can be recognized quickly and efficiently. Wüerges' thesis is the only work known to us that prescribes guidelines for building systems whose entire system design is predicated on a systematic approach to UE-handling throughout all system components. No military or commercial systems presently take this approach. We intend to experimentally evaluate the usefulness of this approach, using the modular computer system described above. The concepts described in Wüerges' thesis and the description of his experiences on a

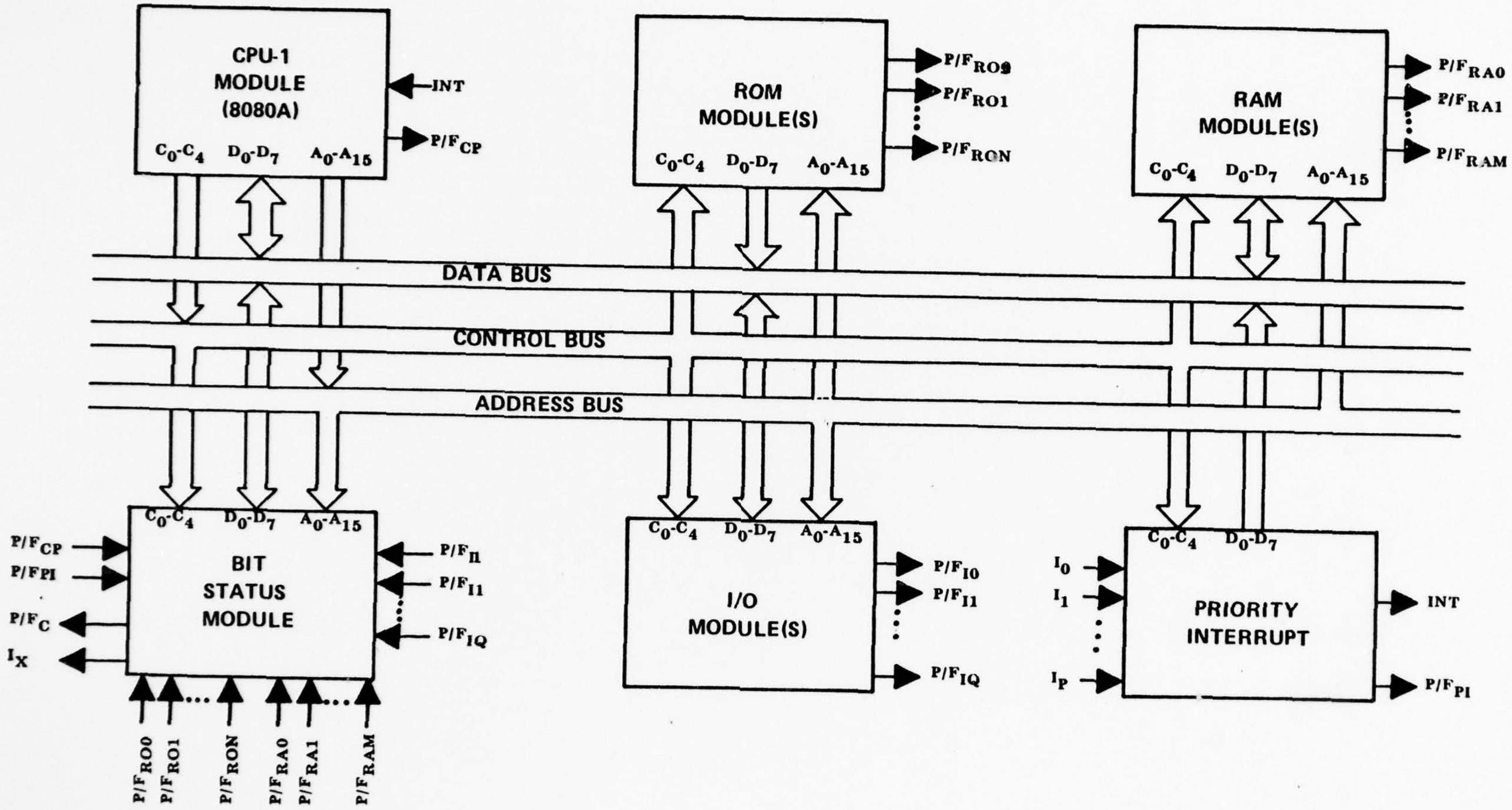


Figure 2. Modular Digital Computer w/Built-In-Test (BIT)

Siemens machine have provided the basis for this research. Past experience has shown that the addition of UE-handling tools provides results (with respect to system reliability) that are comparable to or better than relatively similar effort expended in some other manner. We expect to provide a simulation of the system described above, test results, and generally useful guidelines for designing reliable systems, showing their implementation by means of the model system.

REFERENCES

[Horning 74] Horning, J. J., H. C. Lauer, M. Melliar-Smith and B. Randell. "A Program Structure for Error Detection and Recovery." Colloques IRIA, Rocquencourt, April 23-25, 1974.

[Kaiser 74] Kaiser, C. and S. Krakowiak. "An Analysis of Some Run-Time Errors in an Operating System." Colloques IRIA, Rocquencourt, April 23-25, 1974.

[Parnas 72b] Parnas, D. L. "On the Criteria to be Used in Decomposing a System into Modules." Communications of the Association of Computing Machinery, Vol. 15, No. 12, December 1972.

[Parnas 76a] Parnas, D. L. "Some Hypotheses About the 'Uses'-Hierarchy for Operating Systems." Forschungsber. B5 I 76/1, Technische Hochschule Darmstadt, FB Informatik, February 1976.

[Parnas 76c] Parnas, D. L., G. Handzel and H. Wüerges. "Design and Specification of the Minimal Subset of an Operating System Family." IEEE Transactions on Software Engineering, Vol. SE-2, No. 4., December 1976.

[Parnas 76d] Parnas, D. L. and H. Wüerges. "Response to Undesired Events in Software Systems." Second International Conference on Software Engineering, San Francisco, California, October 13-15, 1976.

[Price 73] Price, W. R. "Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems." Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, June 1973.

[Randell 75b] Randell, B. Private communication.

[Wuerges 76a] Bartussek, W. and H. Wüerges. "Proving that an Implementation Meets its Abstract Specification." Forschungsbericht BSI 76/2, Technische Hochschule Darmstadt, FB Informatik, June 1976.

[Wuerges 76b] Wüerges, H. "Das minimale Teilsystem von BSF: Entwurf, Realisierung and Beweismethode." Vortrag bei der Siemens AG Muenchen, August 1976.

[Wuerges 77] Wüerges, H. Reaktion auf unerwuenschte Ereignisse in hierarchisch strukturierten Software-Systemen. Doctoral Thesis, Technische Hochschule Darmstadt, Federal Republic of Germany, November 1977.

2.3.1 REACTION TO UNDESIRED EVENTS IN HIERARCHICALLY STRUCTURED SOFTWARE SYSTEMS

Harald Wüerges

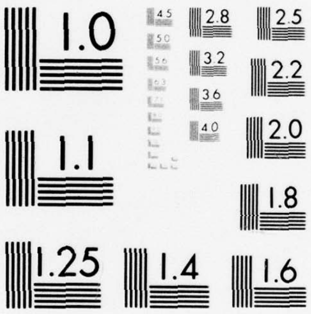
(Translated from German)

Presented as partial fulfillment of the requirements for the awarding of a graduate degree

Technische Hochschule Darmstadt
Federal Republic of Germany

AD-A063 799 RESEARCH TRIANGLE INST RESEARCH TRIANGLE PARK NC SYST--ETC F/G 9/2
BASIC RESEARCH IN SUPPORT OF CONCURRENT FAULT MONITORING IN MOD--ETC(U)
JUN 78 J W GAULT, P N MARINOS, K S TRIVEDI N00039-77-C-0363
UNCLASSIFIED RTI/1504/00-01-1 NL

2 OF 4
AD
A063799



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

CHAPTER 3

REQUIREMENTS FOR THE UE-HANDLING

Structural Aspects

In most programming and system implementation languages, error-handling (if at all practical) is possible only by incurring undesirable program complexity. The presence of all possible UEs must be checked in the program and reaction to these UEs must result in normal program termination. If the number of possible UEs is very large, as, for example, in the case of I/O devices or address translation (virtual to real), then very complex and unclear (i.e., hard to decipher) programs result. This means that changes in the normal code or in the UE-handling are hardly possible. At least in principle, making changes essentially increases the danger of introducing new errors. Since error-handling of the problems here is already difficult enough, an easy surveyability must be guaranteed; otherwise, there exists the danger that error-handling introduces more errors than are treated (i.e., remedied or bypassed).

A separation between code for the normal case and code for the error case provides for the ability to oversee both parts and allows independent changing of both parts. Since experience in dealing with UEs has caused UE-handling to become more comprehensive, this has a special significance. An approach to UE-handling should support this separation and thereby reduce the complexity of the entire program.

A second important point concerns the responsibility for UE-handling. The reaction to a UE in a program depends on the objectives of this program and on the effects that the occurring UE has on these objectives. Therefore, each group of programmers that writes a program for a particular abstract machine should prepare code for the case where their program has errors or where the abstract machine that is used is unable for some reason to execute this program. Only these programmers know what their program was intended to do and, therefore, what actions are possible and sensible for handling this UE. Other programmers know either nothing at all about this program or they know only its specifications, and not its effects. They can also, therefore, not determine what actions should be chosen.

Programs for managing a virtual memory, for example, do not know what is in a segment and for what purpose this segment is used. Whether the content of the segment can be redetermined, or what effects the loss of a segment has, is known only to the user who produced this segment. In a system in which the user himself cannot react to UEs in his program, there remains only the possibility of termination of the program.

The programmers of one level are also most easily in a position to provide information to higher levels, when they themselves are not responsible for an occurring UE or when they see no possibility of a reaction to it.

Both of the user's programs, that for the normal case (the desired case) as well as for the UE-handling, should use the same abstract operations and operands, i.e., the same abstract machine. This guarantees that the programmer uses no knowledge about the implementation of other parts of the system (for example, other modules or submodules); thus, he can make no assumptions about the behavior of other programs that can be changed easily, when changes then result in "incorrect error-handling." This does not mean that the UE-handling uses exactly the same operations as the normal case program; it can restrict itself to a combination and thereby reduce the probability of failure.

This requirement also says that both programs, UE-handling and normal case program, run in the same "environment," i.e., they have access to the same data. It must be guaranteed that no user of an abstract machine can extend his access rights by means of UEs or obtain additional information which would otherwise not be accessible to him (See also Chapter 13.).

Run-Time Behavior

While the last section deals with the structural aspects of UE-handling, I will now consider some requirements for run-time behavior.

With all mechanisms for error-handling, the cost increases with the frequency of UEs[1]. This proportional cost factor is, with many mechanisms (e.g., with "recovery block" mechanism), small compared to the part which is always necessary; thus, also when no UE occurs. (I will discuss the "recovery block" concept more thoroughly later.) [3][4].

In systems without UE-handling, at least equivalent costs arise from system breakdown, loss of data, etc. It is the aim of the concept considered here to minimize the run-time costs which originate from the mechanism and which occur independently of UEs. This means: the cost is small as long as no UE occurs, and, after the UE-handling (successful or unsuccessful), normal processing can be continued as soon as possible (i.e., there result no unnecessary delays because of termination of the processing and because of repetition of already performed actions).

To reduce the costs when the frequency of UEs is great, I see essentially two approaches:

1. One provides for the earliest possible detection of UEs. In this case, a quick reaction to such a UE is possible, and the effect of the UE on the rest of the system can be restricted.
2. By means of timely preparation for possible UEs and by means of sufficient redundancy of data and programs in a system, an occurring UE can be more easily remedied. If, for example, the data on the drum or disk is frequently lost, then the cost of a UE to the user of the system can be kept low by means of periodic copies. This, however, is based on the assumption that the costs of producing copies are less than the costs which are associated with the loss of the data (which is usually the case) (See also Chapter 11.).

There are two important points regarding efficiency with respect to the UE-handling itself (On both points I will expound more completely later. They should only be mentioned here briefly.). Firstly, sufficient information about the occurring UE must be provided. Without such information, reaction to the UE is not possible or only possible with great difficulty. This information includes data about the type of UE, the state of the abstract machine and the possibilities for further processing. This information must exist in a form which is comprehensible to the programmer of the UE-handling.

Secondly, the data related to an occurring UE should be ascertained where it is simplest (cheapest) to do so. Then, the analysis and handling of hardware-detected UEs frequently can be made easier by means of the availability of additional bits. The hardware of the Siemens 4004/151 shows that this is not always the case. There, various UEs are combined under one code, without making available further information for distinguishing the individual UEs (although this information is present in the hardware). (See Chapters 6 and 14.). Distinguishing the individual UEs at higher levels (above the hardware) is only possible at very great expense. This can be avoided if the lower levels (in this example, the hardware) provide the data that are already present or easily obtained; i.e., if the lower levels would make this data accessible.

NOTE (Chapter 3)

1. This affects the processor time as well as all other operating resources required at run-time (e.g., additional memory for data and additional devices). The memory requirement for the code of UE-routines is generally independent of the frequency of the UEs.

CHAPTER 7

THE REPORTING OF UE'S TO HIGHER LEVELS

In Chapter 3, I required that: 1) each programmer provide additional code for the reaction to UEs, and 2) that this additional code use the same abstract operations and operands as the normal case. In the UE-handling of one level, no knowledge about higher levels or the implementation of programs of other modules or submodules may be used.

On the other hand, however, a greater knowledge is generally required for sensible UE-handling than is available at one level. For example, the hardware identifies an improper entry in the address-translation tables; the programs that manage these tables know to which segment this entry belongs; the user of the virtual memory knows which data are in the segment and what the response to the loss is (Further examples of such combinations of knowledge are contained in [5].). Without these bits of information, only general, often drastic measures are possible. Thus, many systems (including the Siemens 4004/151) terminate execution of the processes involved at the occurrence of such a UE.

The necessary combinations of knowledge can be achieved by two different methods: 1) by means of a central routine, and 2) by means of the reporting of an occurring UE between the levels and modules of the system.

The use of a central routine that combines all the necessary information and to which all UEs are reported has some serious drawbacks. Firstly, the modular structure of the system according to the information-hiding principle [2] would be destroyed. This routine must combine the information of several of the system's modules. Each change in one of the modules would necessitate a change in this routine. Secondly, this routine must be able to access all the data of several modules. It would be impossible to protect other programs and data from this routine. This is much more serious, since this central routine will become very complex and, thereby, prone to errors. These disadvantages should surely be avoided by requiring that each programmer prepare code for the event that his program or the abstract machine used fails. If a UE is reported between levels, then each programmer can call upon his knowledge in order to determine the proper measures necessary for the handling of this UE; if the reported UE cannot be handled at one level, it can be reported up to the next highest level.

Two methods are available for the reporting of a UE to higher levels. The first method involves the use of termination code. Each called program has at least one returned parameter. This indicates, after termination of the program, whether an error (and, if necessary, which one) has occurred during execution of the program. The calling program can check this parameter and initiate appropriate actions.

This mechanism, however, does not meet the requirements that were described in Chapter 3 in a concept of UE-handling. This mechanism requires that the user of a program check the returned parameter after each call (in case he does not want to take into account the possibility that UEs remain undetected). Such a check involves an additional expense that is not acceptable when the frequency of errors is low, and in addition, it renders more difficult the desired separation between normal program and UE-handling. Too, this check can be easily forgotten, which leads to UEs remaining undetected. This mechanism is based on the premise that each used program is also called. In Chapter 2 [6], it was shown that "uses" and "calls" do not always have the same meaning.

Another mechanism that also meets the stated requirements depends on the use of traps, analogous to the reporting of UEs by the hardware[1].

"Application conditions" are defined for each program at a given level or for each operation of an abstract machine; these conditions must be met, so that this program can have the specified effect (compare to Chapter 12). Each abstract machine has the responsibility for recognizing all violations of the applicability conditions for one of its operations. In the event of such a violation, control is transferred to a user-defined UE-routine with the aid of traps. This technique makes possible the desired (required) separation of normal program and UE-handling. The user of an abstract machine does not need to make any checks for such UEs in his program. This simplifies the program and reduces the probability that errors go undetected. An additional consequence of this is that, in general, user errors can be detected in the action in which they were produced[2].

Use of the Trap-Mechanism

Each level which is informed of a user error checks the specifications to determine if it, itself, caused this error or if a higher level is responsible for it. A sensible handling of the UE is not possible at intermediate levels; only the level in which it was caused has enough information to take the appropriate actions to handle this UE. For example, only the program that has attempted to read a segment that does not exist knows what should be done with the data that was to be read and what should now be done in the event of error.

Also, if no sensible UE-handling (remedy or bypassing of the UE) is possible at the level in which the UE was caused, then the UE can be reported as a "defect" (in case a further level exists). Termination of the program is always initiated only at the highest level, i.e., at the level of the ultimate user.

Defects depend on the failure of hardware or software components, or on an error by the operating personnel. If an abstract machine is informed of a defect, then it can attempt to remedy it. If this is unsuccessful or if the resources and information available at this level are insufficient, then the defect is reported to the next highest level.

Concerning this passing of information about UEs, two points are noteworthy: 1) the report to the next highest level must be adapted to the abstraction of this level, i.e., the report may not refer to any information that is not known to the programs of this level (Thus, the reports from programs of a virtual memory mechanism to its user may not refer to real addresses.), and 2) at each level, an attempt must be made to lead the appropriate abstract machine into a "possible" state[3]. No user of an abstract machine should receive control, if the machine that is used finds itself in a state in which the relationships between the operations and operands of the abstract machine are not valid[4]. Otherwise, new UEs probably can and will arise, UEs whose cause then becomes considerably more difficult to determine. Krakowiak and Kaiser [7] describe such an error. It appears in conjunction with the synchronization of parallel processes. The error mentioned there can be described as follows, in a simplified manner and with respect to the terminology used here: Part of an abstract machine is implemented with the aid of a critical section, i.e., without parallelism. A UE occurs in the midst of this program segment. If the UE were now reported back to the user, who knows nothing about this synchronization, without first stopping the critical segment (i.e., parallelism is again not allowed), then certainly new errors would arise. Thus, it is absolutely necessary that the abstract machines be put, before the transfer of control, into a state in which all relations are valid, relations which are specifiable for the machine and provable with the aid of the specifications. If a UE is so catastrophic that this transfer is not possible, or possible only for a portion of the abstract machine, then this must be reported to the user. Each further use of this abstract machine must then be prevented, or no responsibility can be assumed for the consequences of such a use.

Avoiding Redundant Error-Checking

An abstract machine can also delegate the responsibility for the recognition of user errors to lower levels. If, for example, a parameter is passed through several levels to a program of a lower level, then one can avoid redundant error-checking by checking this parameter only at the lowest level. If an error is determined there, then the trap mechanism is used in order to report this UE back to the level at which it was caused. In this manner, the cost can be reduced, as long as no UE occurs. If, however, the probability of UEs is very large, then such an economization is no longer possible. The cost for the reporting back of a UE has, then, a strong impact. It would be more favorable, in this case, to check the validity of the applicability conditions at each level, in order to permit fast detection of UEs.

Generally, one will check all applicability conditions in the early phases of the use of a system; if UEs occur only seldom, then some checks can be removed.

NOTES (Chapter 7)

1. On most available hardware machines, the normal execution sequence is interrupted and control is transferred to a predetermined location (chosen by the user or by the hardware) at the occurrence of a UE.
2. If a user error is not immediately detected, and if, therefore, this leads to an incorrect state in the abstract machine, then it will later be detected as a defect of the abstract machine.
3. One can specify for each module or submodule a set of relations which must always be met. One can prove these relations, in that the specifications are considered as a group of axioms. Since each abstract machine is composed of a combination of submodules, I will designate a state of the machine in which all relations hold as a "possible state"; a state in which these relations are not valid for the entire machine or for a part of the machine is accordingly designated as an "impossible" state (For more details, especially of the proof, refer to [8].).
4. It is noteworthy that these invariant predicates are not identical to the specified behavior, but, rather, only secure the contents of the state description. They are therefore not fulfillable if the desired behavior can no longer be achieved. So, for example, a length and access rights must be defined for each existing segment. If this segment description is destroyed, then the validity of this predicate can be restored by designating this segment as "non-existent."

CHAPTER 12

HARDWARE/SOFTWARE INTERFACE

In this chapter, principles for the handling of UEs (undesired events) at the interface between the hardware and software are discussed. They are applied to the interfaces of existing machines (the Siemens 4004/151 and also, in less detail, the IBM/360, IBM/370 and PDP 11). Based on this discussion, we make recommendations for future hardware and software interfaces to offer better support of UE-handling.

In support of these recommendations, several examples taken from the application of the concepts in the implementation of a minimal subset of the BSF (an operating system being developed in Darmstadt, West Germany) on a Siemens 4004/151 are cited. Details of our experience with this implementation are given in Chapter 13.

Undesired Events

In order to allow UE-handling by the user of an abstract machine, he must know the UEs that are possible. With regard to the hardware, this means that all applicability conditions for the operation of a real machine and all the externally distinguishable hardware errors must be explicitly described. What, then, are the possible UEs in a hardware machine? In general, all events that require a special action (an undesired action that is not necessary in a normal program run) are considered UEs.

Errors in individual switching circuits, wires, etc. (e.g., parity errors) and power failure comprise one group of UEs. These are defects in a real machine.

Among the user errors are: improper use of arithmetic operations, the use of undefined or forbidden operations, the specification of an undefined or improperly aligned address, etc. If the real machine has a memory-protection mechanism or address-translation hardware, then additional applicability conditions apply to the particular operations; the violation of those restrictions also represents a UE. The hardware is responsible for the recognition of such a violation of the applicability conditions and for the reporting of a detected UE to the user.

To allow and to support user reaction to UEs, the hardware/software interface should contain a clear (precise) specification of the effect of the available operations. All events which prevent the execution of a specified action should be treated as UEs and reported by means of traps.

A typical example of UEs that are not reported on most machines by the use of traps are I/O errors. Another example is fixed-point overflow on the PDP 11. In both cases, the appearance of a UE is noted by special values in the channel registers or in a program status register. As a result, the user himself must check for the occurrence of a UE in his own program. Firstly, this increases the complexity of the program; secondly, the probability that UEs go undetected is increased. Further, the checking results in additional costs that are also incurred in normal situations.

Classes of UEs

As mentioned in the previous chapter, it is sensible (for efficiency reasons) if the UEs of an abstract machine are grouped into classes for the purpose of reporting their occurrence to the user. The number of classes and the inclusion of a particular UE in one of the classes is dependent upon the interface between abstract machine and user. No information about higher levels and the interface between these levels may be used.

Consequently, the classification of hardware-detected UEs should depend only on the hardware/software interface, and not on an interface assumed to exist between operating system and user.

Below is a classification of UEs in the central processor (I/O errors are strongly dependent upon the available devices; therefore, a generally valid classification would be difficult to construct.). If one considers contemporary interfaces, then one can distinguish the following nine classes of hardware-detected UEs. The first seven classes summarize user errors for a hardware function. Classes eight and nine contain hardware defects.

1. Improper addressing: The specified operation cannot be executed with the operand address (real or virtual) that is specified. For example: improper alignment, an address that is outside of core memory, or improper register pair.
2. Errors in address translation: The virtual operand or instruction address cannot be translated. This class is peculiar to machines that have address-translation hardware. Examples of UEs in this class: inconsistent entries in the translation table; the addressed table entry is not initialized; or the location specified is not in main memory.
3. Data Protection Errors: The attempted access of a data element is not allowed by the defined access rights. This class of UEs depends upon the access rights that are distinguished in the hardware.
4. Unimplemented Operations: The specified operation does not exist.
5. Data Errors in Decimal Operations: An operand in a decimal operation is incorrect (e.g., improper overlap).
6. UEs in Fixed-Point Operations: For example: overflow, underflow, or division errors.
7. Floating-Point Errors: Exponent overflow and underflow; a mantissa which is approximately equal to zero.
8. Hardware Errors: Defects of the hardware, such as parity errors.
9. Power Failure.

This classification is not complete; i.e., not all of the UEs found in existing machines can be put into one of the above categories. At times, additional classes are required, depending upon the complexity of the machine under consideration (e.g., UEs for one of the hardware-handled stacks and emulator errors). The above list gives only the most important classes which can be defined for most of the currently available computers.

Classification Schemes in Existing Machines

If one considers existing hardware/software interfaces, then one generally finds another classification: The Siemens 4004/151, which has address-translation hardware, distinguishes six different traps for UEs arising from arithmetic operations, while all UEs in addressing, address translation and data protection and some other UEs (e.g., emulator-trap) are divided into two UE-groups. These two groups are further subdivided into: 1) user addressing-error, 2) system addressing-error, 3) user address translation errors, 4) system address-translation errors, and 5) memory-protection errors. Here, system refers to the operating system.

Such a "classification" supports reaction to fixed- and floating-point arithmetic, but renders more difficult the analysis and handling of the remaining UEs[2]. This occurs also in the implementation of the minimal subset of BSF [9]. The difficulties arising from this implementation are discussed in Chapter 14.

Division Between Normal Program and UE-Handling

Most hardware machines have an interrupt mechanism by means of which the currently executing command sequence in the central processor can be interrupted, followed by a branch to a predetermined address. An interruption of this type can be caused by four groups of events: 1) synchronous UEs, i.e., UEs which occur at the time of execution of an instruction (e.g., overflow)[3]; 2) normal synchronous (i.e., caused by the currently executing program) events (SVC); 3) normal asynchronous events (e.g., termination of an I/O operation at the request of the console operator); and 4) asynchronous UEs (e.g., I/O errors). Groups (2) and (3) include normal (desired) events; reaction to these events is part of normal processing and should therefore be separate from the reaction to UEs[4].

On most currently used machines, the same mechanism is used for all of these events; i.e., a deviation from the processing sequence is forced and control is switched to a routine specified previously. The address of this routine may be taken from a register or may be a fixed place in memory. The number of distinct routines varies from machine to machine.

On the Siemens 4004/151, only one routine can be defined. All interruptions transfer control to this routine. On the IBM/360, separate routines can be defined for classes (1) and (2),; however, for groups (3) and (4), only one routine definition is assigned. This means that a separation between a normal program (here, the handling of events in groups (2) and (3)) and UE-handling cannot always be maintained. When this separation is possible, it can only be realized with the help of a virtual machine. Thus, additional software is required in order to undo the combination of the two classes by the hardware. This results in the following requirement for hardware/software interfaces: The hardware/ software interface should at least allow for a separate routine for each of the four classes of events. This can be achieved by a minimum of four separate registers or memory locations, in which the addresses for these routines are kept.

Number of UE-Routines

For the separation of normal program and UE-handling, it is sufficient that a distinct routine be defined for each of these groups. The question is whether, for example, restricting the handling of all synchronous UEs to a single trap-routine is possible or sensible. The answer to this question depends heavily on costs and the complexity of the hardware. For a machine that only allows fixed-point arithmetic, a trap-routine is sufficient. But what happens when the machine has fixed- and floating-point arithmetic, address-translation hardware, etc.?

In general, the UE-routines are simpler when they handle fewer UEs. In the extreme case, one would prepare a separate routine for each UE. This, however, is unsuitable for two reasons: (1) In many cases, the number of UEs is too large. The expense involved in the definition of the UE-routines would be very high and would be especially inappropriate for small, inexpensive machines. (2) Such a solution would frequently lead to unnecessary redundant coding, since, in many cases, the same or similar actions are necessary (e.g., the saving of certain registers, similar steps). A feasible compromise exists--grouping related UEs together and handling them by means of a general UE-routine. A classification of synchronous UEs for currently used machines has already been given. A runtime parameter can be used to designate individual UEs within a group. With this in mind, the hardware/software interface should set up, for each group, a register or memory location which contains the address of the UE-routine for this particular class.

Delayed Traps

If processors are used jointly by several processes (successively), and if the processor can, due to the occurrence of an asynchronous event (end of time-slice, termination of an I/O operation, request for a device), be withdrawn from the currently running program and assigned to another, a further problem arises. Synchronous UEs should be handled by the process in which they occur. A process should certainly not be delayed by UEs that do not concern it and which it is not prepared to handle.

Let us consider an example: Assume that an address-translation error caused a trap. Successful handling of the error depends upon the availability of the translation table which led to the occurrence of the UE. If these tables were exchanged in the meantime (e.g., by a process change), analysis and handling are made very difficult.

If one considers current hardware/software interfaces, both synchronous and asynchronous events are indicated by the setting of a bit in an interrupt register. Which of the events leads to an interruption (i.e., causes the transfer of control to a certain routine) depends upon the actual interrupt mask and on the priorities of the individual interrupts. If asynchronous events have higher priorities than synchronous ones, as, for example, on the Siemens 4004/151, it is possible that the reporting of a synchronous UE will be delayed until the handling of a simultaneously occurring asynchronous UE is completed. If the latter contains a process change, which is frequently the case, the UE is reported in the wrong process[5]. To avoid this, one must consider delayed synchronous events as part of the process data and exchange them in a process change. In most currently used machines, this requires a reloading of a part of the interrupt register; the actual contents of part of this register must be stored away with the other data (e.g., registers) of the unloaded process and the new contents gathered from the process data of the process being loaded. Also, the additional information about an occurring UE which is stored in registers must be saved and reloaded.

What effect does such an interface have on the system? (1) The program for the process change becomes more complicated and more costly.

(The storing away and the setting of particular bits in a register is in general very expensive and requires great care on the part of the programmer.)[6]. (2) For the process change, a detailed knowledge of possible traps and the registers used for provision of run-time information is necessary.

To avoid these difficulties, the hardware/software interface should either assign traps higher priority than interrupts or support the storage of the (information absent) delayed traps and UE-information, while providing operations for the storing and retrieval of this data. It would be best, therefore, if the reaction to synchronous UEs (precisely, the part of the reaction which requires a ban on interrupts) were as short as possible, so that asynchronous UEs do not have to be delayed too long.

"Environment" for UE-Handling

For abstract machines, I required that each programmer (or each group of programmers) who writes a program for this machine prepare additional code for the case where the original program has errors or where the abstract machine is, for some reason, not able to execute this program. This UE-handling should use the same abstract operations and data structures as the original program; i.e., it should work in the same "environment." The user is not allowed, by means of UE-handling, to obtain additional information that is normally unknown or protected.

For the hardware machine, this means that UE-handling by the user should have the same privileges and the same access rights as the user's normal program. This environment must be produced by the occurrence of a UE. The associated information must be already stored in special registers or memory locations. The hardware/software interface should support the explicit definition of the environment for each interrupt routine and the automatic production of this environment at the occurrence of the corresponding interrupt. In addition to the register or memory location for the address, for each group of interrupts, additional registers or memory locations, whose contents indicate the environment for the appropriate interrupt routine, should be available.

The example of the minimal subset of BSF [9][10][11] shows that it is not sufficient for the values of the control register to be retained at the occurrence (or detection) of a UE. This minimal subset makes available a virtual memory and the operations for this virtual memory. Since the Siemens 4004/151 has address-translation hardware, no software is necessary for address translation. The control registers thus always indicate the environment of the actual user of the minimal subset, not the environment in which the address translation exists. The reaction of the minimal subset to hardware-detected UEs must, however, be implemented by software routines. These routines must work with real addresses and have access to the translation table. Thus, the production of the necessary environment must be done explicitly, i.e., by means of software.

As already mentioned, the hardware of the Siemens 4004/151 transfers control to the same routine for synchronous and asynchronous events; this routine works in a particular program state. The result of this is that these general routines: 1) function with real addresses (interrupt handling can also function with virtual addresses, and 2) must have access to all data which are necessary either for UE-handling or for interrupt-handling. Because of these maximal access rights, this routine represents one of the critical parts of the system[7].

If the proposed additional registers or memory locations were provided, the environment necessary for each routine could be produced directly from the hardware. A special software routine with maximal access rights would not then be necessary.

Relationship Between UE and UE-Handling

Each abstract machine can be used by several different programs at higher levels. Each "user" can have his own UE-handling. As discussed in Chapter 6, the actual user's provision and the report of the UE to this user can be supported by a dynamic relationship between UE and UE-routine. In hardware machines, the relationship between UE and UE-routine can be changed, when one changes the trap-address (i.e., the address of the trap routine) at run-time.

Changing of the trap address should be supported by the hardware so that no knowledge of other parts of the system (e.g., interrupt-handling) is necessary for this change. In most currently available hardware/software interfaces, the above feature is not included. In the Siemens 4004/151, the changing of the trap address means simultaneously changing the interrupt address (i.e., it defines a new interrupt routine). On the IBM 360 and the PDP 11, the trap address can indeed be changed without influencing the handling of other interrupts; however, the trap addresses and the addresses of the interrupt-handling routines are in the same memory area. The access rights which are necessary for the changing of the trap address also allow access to the addresses of the other interrupt-handling routines. If part of the data should be changed by a program, it is not possible to protect the rest of the data against unauthorized changes by the program.

In order to support the information-hiding principle [2] and for data protection among programs, the hardware/software interface should provide separate data areas with individual or separate access rights that can be specified singly.

Resumption of Normal Execution

In Chapter 5, I introduced three functions which should be available to the user of an abstract machine for the resumption of normal execution: CONTINUE, RETRY and CLEAR.

If one considers currently used hardware/software interfaces, none offers a corresponding set of functions. The only way to implement the functions CONTINUE and RETRY is to explicitly reset the old (saved) value of the instruction counter (i.e., the value at the time that the UE occurred) in the actual instruction counter (by loading the appropriate register). All of these actions must be performed by software. In most cases, there are, in addition to the instruction counter, various other control registers to load, in order to recreate the environment of the interrupted program. The only machine known to me which allows one to take control of the instruction counter is the PDP 11. The hardware/software interface of this machine offers an operation which exchanges the old stored value of the instruction counter with the actual value of the instruction counter and thereby allows the resumption of the interrupted program. This represents an important feature of the software.

Hardware/software interfaces should supply a special mechanism (instruction) for the resumption of an instruction sequence which has been interrupted by a trap.

An essential difference between general abstract machines and hardware machines is that resumption of an interrupted hardware operation by the hardware is not usually possible. Before the notification of the presence of a UE to the user (i.e., before the occurrence of a trap), the currently executing operation is terminated; continuation is possible only with a new operation. This property is certainly efficient, when it handles, as is the case with most hardware commands, simple operations with short execution time. The situation is different when a machine performs complex operations. Let us consider, for example, a machine with address-translation hardware. In such a machine, command execution is carried out in two parts: 1) translation of the virtual address into a real one; 2) execution of the operation with the operands which are designated by the real addresses. In most machines, a UE in one of the two parts leads to termination of the currently executing instruction and to the occurrence of the trap.

Normally, no memory location or register is changed during address translation (except some that are internally used and not externally noticeable). After removing the cause of the UE, it would be possible in principle to continue or repeat the interrupted operation. Such a resumption or repetition of an interrupted operation is not supported in current hardware/software interfaces. The resetting of the instruction counter, and eventually other registers, must be carried out explicitly by the user of the hardware. Here also the PDP 11 shows a step in the right direction. The designers of the hardware recognized the necessity of such a possibility and gave a comprehensive description of the actions required. The programming of these actions is, however, left to the user. Stronger support by the hardware would lead to greater efficiency and to a lower probability of errors.

Hardware/software interfaces should supply mechanisms that can be called upon to continue or repeat a partially completed operation after the cause of the UE is removed.

Defects of Hardware Functions

Some actions of an abstract machine can also be carried out by the user of the machine with the aid of other operations of the same machine. For example, with the help of integer arithmetic in FORTRAN, signal processing can be implemented. This implementation is indeed very inefficient, but it is theoretically possible.

For real machines, similar examples can be found. Floating-point arithmetic can be implemented with the aid of fixed-point arithmetic. This method is less efficient than a hardware implementation, but the results are the same. Address translation presents another interesting example. On a Siemens 4004/151, if the address-translation hardware fails, the entire system collapses. Its function, the translation of virtual addresses into a table of available locations in memory, can also be carried out by a program. These programs would need more time for the translation, but would produce the same results. In order to allow for such a program, there would have to be the possibility of communication of virtual and real addresses between hardware and program. This can happen if the internally used address registers can be made available by special commands or as special memory addresses. Not all hardware components can be replaced by software programs; other hardware components have such a low probability of failure that special preparations for the possibility of failure are not worthwhile. Further, the software implementation will certainly not be considered a long-term alternative. It can, however, bridge over short-term difficulties and permit an orderly shutting down of the system.

If one must contend with an occasional failure of hardware components, possible software implementations of these components should not be excluded. The hardware/software interface should allow for such an implementation and provide for an interchange of results.

Costs of the Proposed Changes

In this section, the proposed changes to the hardware/software interface in the example of the Siemens 4004/151 will be made concrete. At the same time, the costs for an altered interface will be analyzed. The Siemens 4004/151 is equipped with four register sets. Each of these sets consists of the control registers ISR (Interrupt State Register), IMR (Interrupt Mask Register) and the PC (Program Counter), and a set of general purpose registers. In addition, the machine is equipped with a BTAR (Block Table Address Register). The register sets are closely related to the four program states (hardware errors, interrupt analysis, interrupt handling and user program). The number of multipurpose registers and the addressing of the control registers are not the same for all states[8]. At the moment, an interrupt is provided in each channel in the Siemens 4004/151, by means of which both normal termination occurs and the occurrence of an error is reported. This existing interface should be changed as follows:

1. The events of each of the four groups (synchronous normal, synchronous undesired, asynchronous normal, asynchronous undesired) are reported by separate interrupts. Since only two synchronous normal events are possible (SVC and test operation), only two interrupts are required for this group. For each class of synchronous UEs (refer to the recommendations at the beginning of this chapter), a separate interrupt should be provided. In addition to the previously discussed interrupts for normal asynchronous events, separate interrupts for the corresponding asynchronous UEs should also be provided. With respect to the current situation, this would require three additional interrupts[9].
2. The strict association of program states with register sets is abolished. Instead, three or four independent sets of multipurpose registers are proposed. The actual register set is indicated by a bit combination in one of the control registers. The control registers, including BTAR, are necessary only in a model.

For this change, no additional registers are required. Only their strict relationship to program states and special tasks is removed.

3. In the event of an interrupt, the control register is reloaded by the hardware. For each group of interrupts, a separate area of memory is provided. This contains the new values of the control registers. The old values are stored in a memory location whose address is stored in a register or a specially-allocated memory location. In this case, an extension of the mechanism to a definition of separate memory areas for each interrupt incurs only additional memory costs. For the protection of this data against unauthorized accesses, it would be best if separate access rights could be assigned for each of these memory areas. This can be achieved, for example, by a virtual memory which allows the definition of very small segments (several words). The Siemens 4004/151 has an address-translation mechanism; however, the minimum segment size is 4096 bytes. The situation is improved if one allows a change of the data for the interrupt routines (address and environment) only through special, protected programs. The correct method and the protection of this program secures simultaneous protection of the data. The problem before us is that traps which have not yet appeared can be lost if one assigns synchronous events a higher priority than asynchronous events. The costs for this change are low. The routines for reacting to synchronous events must be so laid out that the reaction to asynchronous events is delayed as little as possible.

Two machine commands (CONT and RETRY) can be provided for the resumption of the interrupted execution and the restoration of the interrupted state; CONT picks up execution at the place where it was interrupted (generally, with the next operation in the command sequence) and RETRY repeats the last command[10]. The operands of these operations are that of:
1) the register set being used, and 2) the address of the memory area in which the old values of the control and address registers are stored.

If only one register set is available, the first operand is not used. The cost for these two additional commands is relatively low on most of the large machines.

In order for address translation to be carried out by software, in the event of a defect in the address-translation hardware, the internally used registers should be made available either as special registers or as special words in memory. Upon the occurrence of an address-translation error, the currently executing operation would then be interrupted, not terminated. A resumption of this operation should then be possible with CONT or RETRY. This change might, as previously mentioned, be controversial; however, it allows an orderly shutting down of all system components and eventually some user programs, in the event of address-translation failure.

On other machines, different criteria would be used in considering the costs for the changes cited above. In the case of the Siemens 4004/151, few additional costs arise; essentially, only unfavorable, existing design decisions had to be revised. In contrast, additional registers and commands in small, inexpensive computers can represent a significant cost factor. Therefore, one must analyze in the design of a machine all possibilities for achieving the ideas set forth in this section and determine the cost; in most cases, the final design represents a compromise between the requirements presented and the cost of implementing them.

NOTES (Chapter 12)

1. Since I/O devices are very slow, the costs for the checking (performed mainly by the central processor) of I/O errors is small compared to the determination of errors in the central processor. The costs of further interruptions can sometimes be higher than the costs of error determination supplied by the program.
2. A similar classification exists for the IBM/360, /370.
3. Hereafter, I will use the term "traps" for interruptions which are caused by events in group (1).
4. A separation between synchronous and asynchronous events is desired, since, in general, different users of the hardware machine must react to these events. In addition, for the reaction to synchronous events, a built-in stack can be used; for asynchronous events, the use of a stack is questionable, since, for the sequence of reactions to asynchronous events, factors other than the order of occurrence can be decisive.
5. In general, this problem does not arise in the case of I/O errors, since it is not normally possible to interrupt the currently executing channel program and later continue. The unloading of a channel process occurs only as a reaction to an I/O error in the execution of the process or after the successful completion of the I/O. The multiplex channel is an exception; however, the switching between processes is carried out completely by the channel itself.
6. Note that these actions must be carried out in such a way that no interrupts are lost.
7. These access rights, then, are also required if the general routine implements a virtual machine in which traps and interrupts are separated and have different access rights. In this case, the general routine must have the right to confer any privileges and access rights. It is, therefore, necessary for the routine itself to have these rights.
8. For example, in state P3, the control registers of the other states can be regarded as multi-purpose registers.

9. If one considers fixed-point, floating-point and decimal errors as subclasses of a general class of UEs, one can manage with the previous number of interrupts. Then, only the relationship of the interrupts to the individual events need be changed.

10. On the Siemens 4004/151, RETRY allows only a resetting of the instruction counter; on the PDP 11/45, it is possible that some address registers must also be reset.

CHAPTER 14

COSTS OF UE-HANDLING

The UE-routines of the minimal subset occupy about five hundred 32-bit words (including data). A direct comparison with the memory requirement of the normal program of the minimal subset is not possible, since the address translation is carried out by the hardware. The only software routine of the minimal subset, MAPSELECT, includes twenty commands (the memory requirement for the data depends on the number of address spaces).

The number of instructions executed in the UE-case is between fifteen and eighty machine instructions, depending on which UE occurred. The costs for the production of the UE-routines of the minimal subset totalled approximately two man-months. The first simple versions were completed in two weeks with the aid of a mathematical-technical assistant.

The relatively high costs of the handling of some UEs resulted essentially from the following causes: the classification of the UEs that were reported by the hardware and the information about these UEs were insufficient. The classification came from the interface between a complete operating system and its user. The minimal subset makes available, however, only a simple virtual memory with a fixed number of address spaces and a fixed number of segments. In order to report UEs to the user of the minimal subset, they must be analyzed separately and re-grouped into classes. This was made more difficult, since the hardware provided insufficient or no information about the exact UE in a class.

Some UEs, therefore, could generally not be identified; others had to be determined by the exclusion of all other UEs in their class. An example is the previously mentioned RESTOK-bit; this indicates whether an interrupted instruction is repeatable. Some UEs appear twice within a class in the classification of the hardware, once with a set and once with an unset RESTOK-bit. This bit, which would be necessary in order to distinguish between the two situations, is not accessible, however, outside the hardware.

This detailed and somewhat complicated analysis of occurring UEs contributed substantially to the relatively high costs of UE-handling for the minimal subset. A lowering of the costs is possible in two ways: 1) the hardware/software interface is changed, or 2) one dispenses with a detailed analysis of UEs. One notes that, in the first case, no change to the interface to the user of the minimal subset is necessary. Only the UE-analysis in the UE-routines of the minimal subset must be changed. If one dispenses with a detailed analysis, then the possibilities for UE-handling at the higher levels are thereby restricted. A subsequent definition of an individual UE at higher levels is generally impossible or very expensive[1].

An additional cause for the costs of the UE-handling is the possibility of a resumption of normal processing. Then, the values of the control register (and perhaps some other registers) must be saved at the time of occurrence of a UE and later reloaded. This requires, on the one hand, memory space for these registers (about 10 words) and, also, additional time for the execution of the corresponding instructions (about five instructions). If one dispenses with the possibility of resumption of the interrupted command, i.e., if one permits basically only the use of CLEAR, then these costs can be economized (If one still wants to resume normal execution with some UEs, then, in most cases, this will require a repetition of already successfully executed actions.)[2].

This shows that the costs of UE-handling can also be reduced by the proposed concept, if one reduces the demands on the UE-handling to a minimum (all UEs are reported as one, no possibility of resumption of normal processing). This possibility is not excluded by the proposed method and, more importantly, also does not increase the costs. The least amount of cost which is incurred by implementation of the proposed concept is the cost for the realization of the traps for the reporting of UEs up to the highest level (where termination can then result). These costs can only be avoided then, if from the beginning, each reporting and therefore each handling is dispensed with and if execution halts with the occurrence of a UE at one of the lower levels; but such an assumption must then be made in the entire system and would not then be changed, or only with great expense and difficulties.

NOTES (Chapter 14)

1. For example, the UE-routines of the minimal subset can call upon the address-translation tables for the analysis of UEs and for determination of the affected segment. The programs of higher levels have no access to it. An exception are the programs of higher levels which belong to the same module as the minimal subset.
2. Note that the costs do not depend on whether one considers only CONTINUE or only RETRY as continuation possibilities.

References

1. Wüerges, H., Reaktion auf Unerwuenscnte Ereignisse in Hierarchisch Struktierten Software-Systemen, Doctoral Thesis, Technische Hochschule Darmstadt, Federal Republic of Germany, November 1977.

2. Parnas, D. L., "On the Criteria to be Used in Decomposing a System Into Modules," Communications of the Association of Computing Machinery, Vol. 15, No. 12, December 1972.

3. Horning, J. J., Lauer, H. C., Melliar-Smith, M., Randall, B., "A Program Structure for Error Detection and Recovery," Colloques IRIA, Rocquencourt, April 23-25, 1974.

4. Randell, B., private communication.

5. Parnas, D. L. Wüerges, H., "Response to Undesired Events in Software Systems," Second International Conference on Software Engineering, San Francisco, California, October 13-15, 1976.

6. Parnas, D. L., "Some Hypothesis About the 'Uses'-Hierarchy for Operating Systems," Forschungsber. B5 I 76/1, Technische Hochschule Darmstadt, FB Informatik, February, 1976.

7. Kaiser, C., Krakowiak, S., "An Analysis of Some Run-Time Errors in an Operating System," Colloques IRIA, Rocquencourt, April 23-25, 1974.

8. Price, W. R., "Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems," Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, June, 1973.

9. Parnas, D. L., Handzel, G., Wüerges, H., "Design and Specification of the Minimal Subset of an Operating System Family," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December, 1976.

10. Bartussek, W., Wüerges, H., "Proving That An Implementation Meets Its Abstract Specification," Forschungsbericht BSI 76/2, Technische Hochschule Darmstadt, FB Informatik, June, 1976.

11. Wüerges, H., "Dasminimale Teilsystem von BSF: Entwurf, Realisierung und Beweismethode," Votrag bei der Siemens AG Muenchen, August, 1976.

APPENDIX A

THE BEHAVIOR AND SIMULATION OF RELIABILITY MODELS FOR BUILT-IN-TEST EQUIPMENT

THE BEHAVIOR AND SIMULATION
OF RELIABILITY MODELS FOR
BUILT-IN-TEST EQUIPMENT

by

Peter Joseph Crotty

Department of Computer Science
Duke University

Date: April 27, 1978

Approved:

K S Trivedi
Kishor S. Trivedi, Supervisor

P. N. Marinos

Thesis submitted in partial fulfillment of
the requirements for the degree of Master
of Arts in the Department of
Computer Science in the Graduate School
of Duke University

1978

ACKNOWLEDGEMENTS

This work was supported in part under the Naval Electronics Systems Command, Contract No. N00039-77-C-0363.

The author also wishes to acknowledge the assistance of Professor Kishor S. Trivedi, whose support and guidance was invaluable in the preparation of this thesis.

And finally, the author wishes to express appreciation to his wife, Eleanor J. Crotty, for her encouragement and understanding.

CONTENTS

I. INTRODUCTION

The primary unit under investigation is any integrated-circuit (IC) device with simple functionality. As an example, consider an eight bit full-adder in a DIP ceramic package. A certain reliability will be associated with such a unit. The nature of this reliability reflects the age state of the device.

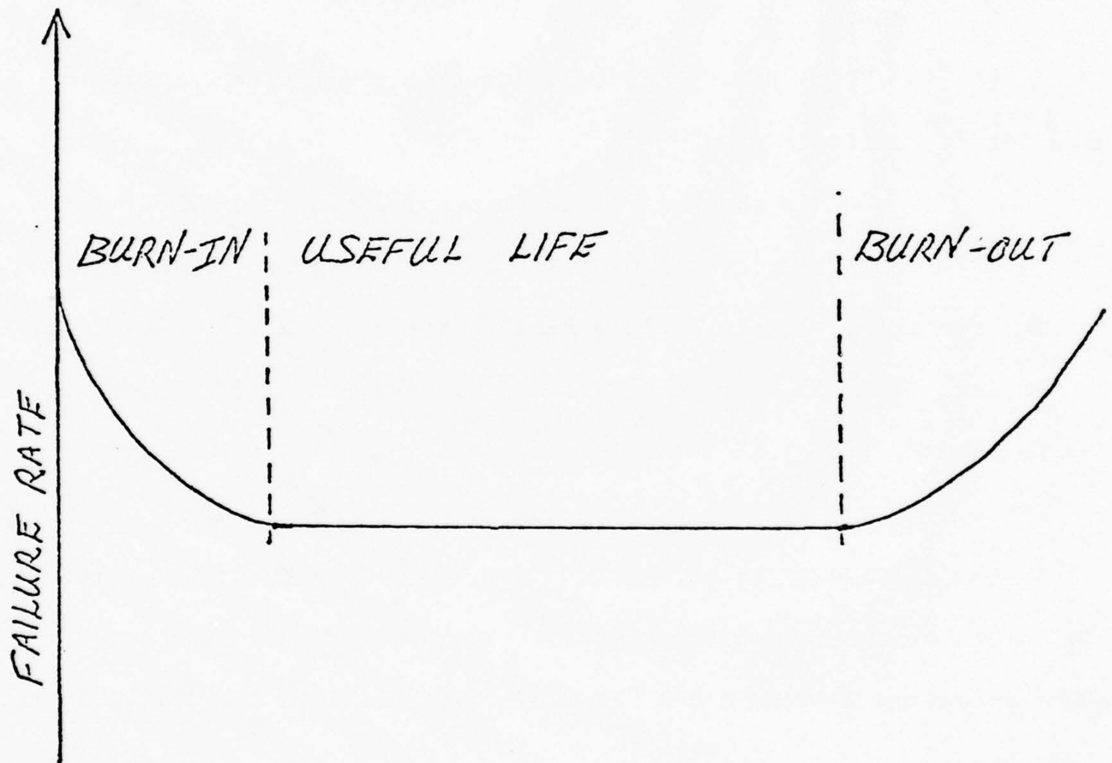
Three fundamental age states are widely used to characterize the reliability [1].

State 1: ("Infant Mortality" or "Burn-In"). When new devices are first brought into operation, they exhibit a very high failure rate. This is due to "bad" units: that is, devices that suffered a major manufacturing flaw, damage during installation, etc.

State 2: ("Useful Life"). After the grossly defective units have been weeded out, the remainder continue to operate at an average failure rate that is acceptably low.

State 3: ("Burn-Out"). After a certain time in operation, the average failure rate increases rapidly. This is simply due to wear and tear on the device, and the eventual deterioration of its components. See Graph 1.

It is the behavior during the "useful life" that is considered in this study.



Graph 1

II. THE MODELS OF THE SYSTEM

1. The Maintained System

The failure of the device is modeled by an exponential distribution [2].

(1) $R(t) = e^{-\lambda t}$

where,

"R(t)" is the probability that the device suffered no failures for "t" units of time.

"e" is the base of the natural logarithm: 2.718.....

"λ" is the failure rate of the device in (FAILURES/HR.).

The reciprocal of the failure rate is the average time until a failure occurs. This is an important quantity, and is called the "mean time to failures", (MTTF) [3].

(2) $MTTF = 1/\lambda$

In the given example of a full adder, a failure could be an instance when two operands are added incorrectly. The system would then be described as having gone from the "working state " (W) to the "failure state" (F). See Fig 1.

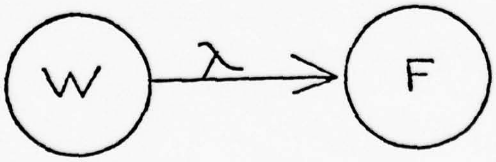


Fig 1

In state "F", the user is relying on a system that is providing erroneous results. In order to quickly notify the user of this condition, and make the necessary repair, a "built-in-test" (BIT) mechanism operates concurrently with the unit.

The BIT equipment might take the form of an additional IC component performing modulo addition on certain bit positions of the operands. It then would compare its result to certain bit position values of the full-adder's result. Alternatively, the BIT equipment might be a software routine.

In any case, the significant characteristic here is that detection is performed by a "checker" that is on-line. The advantages of having such detection capability are obvious, and have been studied: "The purpose of associating a detector with a module is for reducing the propagation and contamination of errors and also for easing the maintenance, since the faulty modules are located automatically by themselves. Therefore the repair time is reduced." [4]

The time it takes the checker to detect a failure is termed the "latency" [8].

This also is modeleld with an exponential distribution.

(3) $L(t) = e^{-\delta t}$

where,

"L(t)" is the probability that detection occurs after "t" units of time.

"δ" is the rate of detection in (DETECTIONS/HR.).

The reciprocal of the detection rate is the average time it takes to detect a failure. This quantity is generally referred to as the "mean time to detect failure" (MTDF).

(4) MTDF = $1/\delta$

The system is now described as having gone from the "failure state" (F) to the "detected state" (D). See Fig 2.

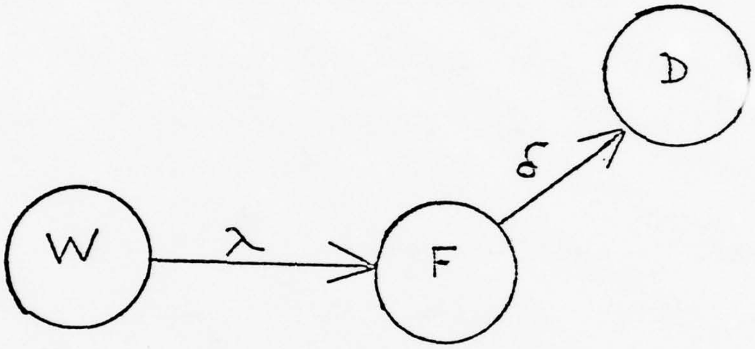


Fig 2

Once the detected state is reached, the necessary repair or replacement is done to return the system to the working state. This regained working state is completely comparable to the previous working state. For example, if replacement were made, the new full-adder chip would have already undergone burn-in and would be in its useful life.

The repair time is also an exponential distribution.

(5) $M(t) = e^{-\mu t}$

where,

"M(t)" is the probability that the repair is made after "t" units of time.

"μ" is the repair rate in (REPAIRS/HR.).

The reciprocal of the repair rate is the average time it takes to make a repair: the "mean time to repair" (MTTR).

(6) $MTTR = 1/\mu$

The system is now depicted in Fig 3 [8].

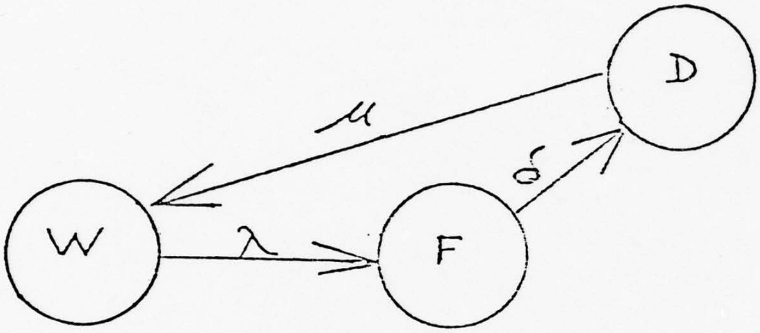


Fig 3

The final event to be considered is the failure of the checker. This probability is given by:

(7) $C(t) = e^{-\alpha t}$

where,

"C(t)" is the probability that a checker failure occurs after "t" units of time.

"α" is the failure rate in (FAILURES/HR.).

The time to repair the checker is modelled by:

(8) $B(t) = e^{-\beta t}$

where,

"B(t)" is the probability that the repair is made after "t" units of time.

"β" is the repair rate in (REPAIRS/HR.)., of the checker.

The complete system is now shown in Fig 4. [5, 8]

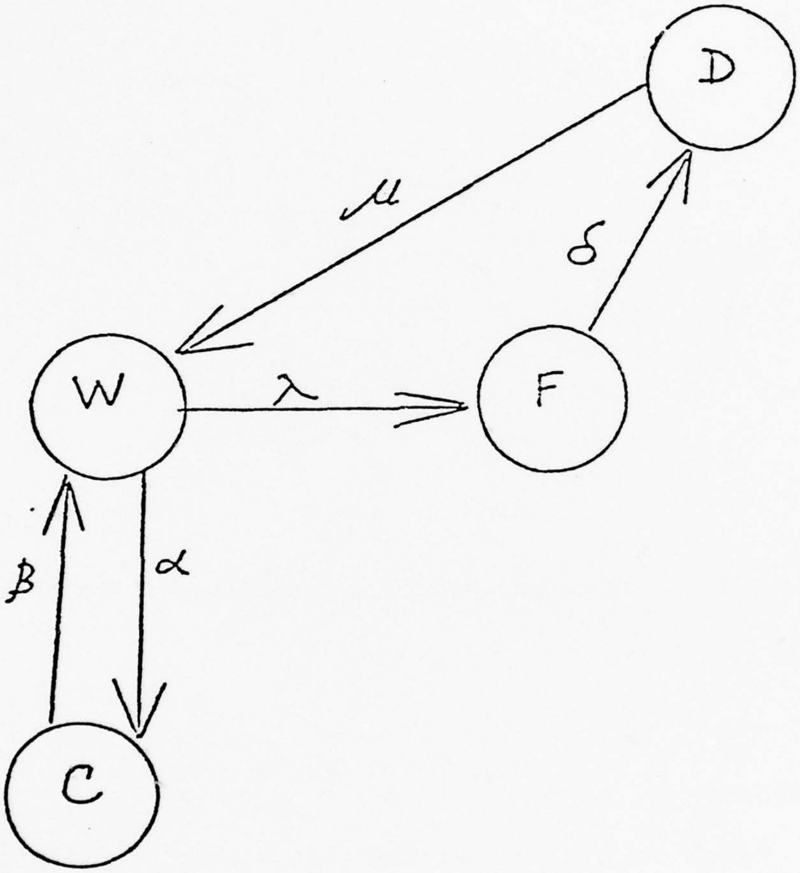


Fig 4

After the system model, depicted by the Markov Chain in Fig 4, has been running a long time, it reaches the "steady-state". [6] Once the steady state is reached, there is very little change in its behavior from one time interval to the next. At this point, the probabilities of being in a given state are derivable. The probability of being in state "W" (P_W) and the probability of being in state "F" (P_F) are of particular interest here:

(9) $$P_W = \frac{1}{1 + \lambda/\mu + \lambda/\delta + \alpha/\beta}$$

(10) $$P_F = \frac{\lambda/\delta}{1 + \lambda/\mu + \lambda/\delta + \alpha/\beta}$$

The "P_W" is called the "real availability" (A_{REAL}). This is the probability that the system is producing correct results. When the unit fails, however, there is a certain latency time before this failure is detected. Before detection occurs, the user assumes the system is available, unaware that it is in the failure state. For this reason, the probability of being in state "W" or state "F" is called the "apparent availability" (A_{APPR}) [5].

(11) $A_{REAL} = P_W$

(12) $A_{APPR} = P_W + P_F$

(13) $A_{APPR} = A_{REAL} + (\lambda/\delta)\ A_{REAL}$

2. The Non-Maintained System

This system consists of a functional unit and a checker. There is no repair done, however, when a unit failure is detected, or when the checker fails.

Let,

$1/\lambda$ = mean time to unit failure

$1/\delta$ = mean time to unit failure detection

$1/\alpha$ = mean time to checker failure

Then, the "real reliability" of the system is given by:

(14) $R_r(t) = e^{-(\lambda+\alpha)t}$

The "apparent reliability" of the system is given by:

(15) $$R_a(t) = \frac{\delta}{\delta-\lambda} e^{-(\lambda+\alpha)t} - \frac{\lambda}{\delta-\lambda} e^{-(\delta+\alpha)t}$$ [5]

And the real and apparent mean times to failure for the system:

(16) $$MTTF_R = \frac{1}{\lambda+\alpha}$$

(17) $$MTTF_A = \frac{\delta + \lambda + \alpha}{(\lambda+\alpha)(\delta+\alpha)}$$ [5]

It is these quantities that are of interest in the non-maintained system.

III. BEHAVIOR OF THE SYSTEM

Choosing values for the equations of interest was a problem with no clear answers. A small (eight bit) adder can be constructed that has a mean time to failure (MTTF) of 10^5 hours. The checker, being a smaller, less complex device, can have an MTTF of 10^6 hours. If, for example, a modulus three bit checker is employed, then hardware detection can be done in 0.01 seconds. Allow 15 minutes to repair the checker and 30 minutes to repair the unit.

Given these parameter values, the availabilities for the maintained system, to seven significant digits, are:

$A_{REAL} = 0.9999947$

$A_{APPR} = 0.9999947$

For the non-maintained system:

$MTTF_R = 90909.09$ hrs.

$MTTF_A = 90909.09$ hrs.

If the detection time is increased tenfold to 0.1 seconds, still no change is registered within this precision (see Tables 1 and 2).

For this reason, double precision (two full words of memory) was required for all values. For producing graphs, extensive scaling had to be done. In general, those digits that remained constant for both the real and apparent cases, over a range of detection times, were subtracted out.

In order to examine how availabilities (maintained) and MTTF's (non-maintained) depend upon latency, five detection time ranges were used.

TABLE 1

```
MTTF - DEVICE  = 1.0E+5 HRS.
MTTF - CHECKER = 1.0E+6 HRS.
MTTR - DEVICE  = 0.5 HRS.
MTTR - CHECKER = 0.25 HRS.
MTDF:  0.01 SEC. -> 0.1 SEC


MTDF                      REAL.AVAIL               APPR AVAIL
 2.777777777777777E-06     9.999947499997851E-01    9.999947500275627E-01
 3.052503052503052E-06     9.999947499970377E-01    9.999947500275627E-01
 3.387533875338753E-06     9.999947499936876E-01    9.999947500275627E-01
 3.805175C38051750E-C6     9.999947499895112E-01    9.999947500275627E-01
 4.340277777777777E-06     9.999947499841603E-01    9.999947500275627E-01
 5.C50505050505049E-06     9.99994749977058CE-01    9.999947500275627E-01
 6.038647342995168E-06     9.999947499671766E-01    9.999947500275628E-01
 7.5075C7507507505E-06     9.999947499524881E-01    9.999947500275628E-01
 9.920634920634920E-06     9.999947499283572E-C1    9.999947500275630E-01
 1.461988304093567E-05     9.999947498813652E-01    9.999947500275633E-01
```

TABLE 2

```
MTTF - DEVICE  = 1.0E+5 HRS.
MTTF - CHECKER = 1.0E+6 HRS.
MTDF:   0.01 SEC. -> 0.1 SEC

MTDF                       MTTF REAL                  MTTF APPR.
 2.777777777777777E-06      9.090909090909089E+04      9.090909091161612E+04
 3.052503052503052E-06      9.090909090909089E+04      9.090909091186588E+04
 3.387533875338753E-06      9.090909090909089E+04      9.090909091217044E+04
 3.805175038051750E-06      9.090909090909089E+04      9.090909091255012E+04
 4.340277777777777E-06      9.090909090909089E+04      9.090909091303657E+04
 5.050505050505049E-06      9.090909090909089E+04      9.090909091368224E+04
 6.038647342995168E-06      9.090909090909089E+04      9.090909091458056E+04
 7.507507507507505E-06      9.090909090909089E+04      9.090909091591589E+04
 9.920634920634920E-06      9.090909090909089E+04      9.090909091810963E+04
 1.461988304093567E-05      9.090909090909089E+04      9.090909092238167E+04
```

MTDF = 0.01 → 0.1 seconds

0.1 → 1.0 seconds

1.0 → 10. seconds

1.0 → 10. minutes

10. → 20 minutes

In order to extend the analysis to more complex units, additional parameter values were used. In all, three cases were examined.

CASE:	I	II	III
MTTF-DEVICE:	10^5	10^4	10^3
MTTF-CHECKER:	10^6	10^5	10^4
MTTR-DEVICE:	0.5	1.0	2.0
MTTR-CHECKER:	0.25	0.5	1.0

The MTTR's apply only to the maintained system. All values are given in hours. In accordance with the checker being a less complex mechanism than the unit device, it is assumed that the checker takes longer to fail and quicker to repair. Each of these three cases was examined over the five specified latency time ranges.

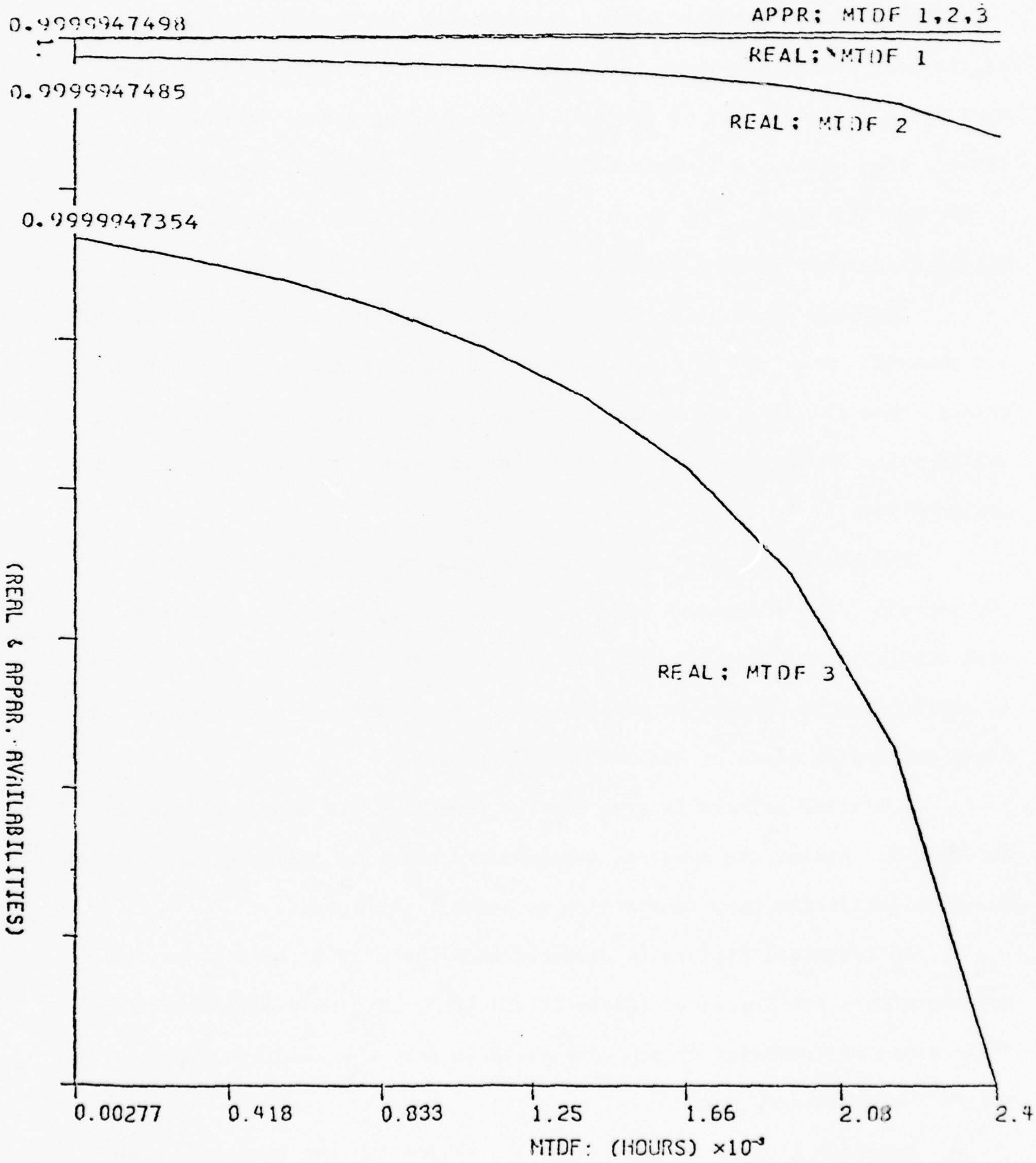
1. The Maintained System

See Graph 2. As indicated in the graph labels, this depicts the real and apparent availabilities over the first three latency ranges.

The straight line at the top (1.0×10^{-8}) represents the apparent availability. Its horizontal character indicates that this value undergoes no change relative to the real availabilities. Of course, as the latency increases, the apparent availability also increases since the user is unaware of a failure for a longer time.

GRAPH 2

MTTF - DEVICE = 1.0E+5 HRS.
MTTF - CHECKER = 1.0E+6 HRS.
MTDF 1: (0.01 -> 0.1) SECS.
MTDF 2: (0.1 -> 1.0) SECS.
MTDF 3: (1. -> 10.) SECS.



This increase can be seen in Table 1, and takes place in the 10^{-13}, 10^{-14}, and 10^{-15} decimal places.

The almost straight line directly below the apparent availability is the real availability for MTDF from 0.01 to 0.1 seconds. The system spends the vast majority of its time in the working state. When a unit failure does occur, it is detected very quickly, repaired, and returns to the working state. Due to this very small detection latency, there is little difference between the real and apparent availabilities.

The next curve below is the real availability for MTDF from 0.1 to 1.0 seconds. When the unit does suffer a failure it now takes longer to detect, thus delaying the return to the working state. Therefore the real availability is decreased, as indicated by the wider gap from the apparent availability.

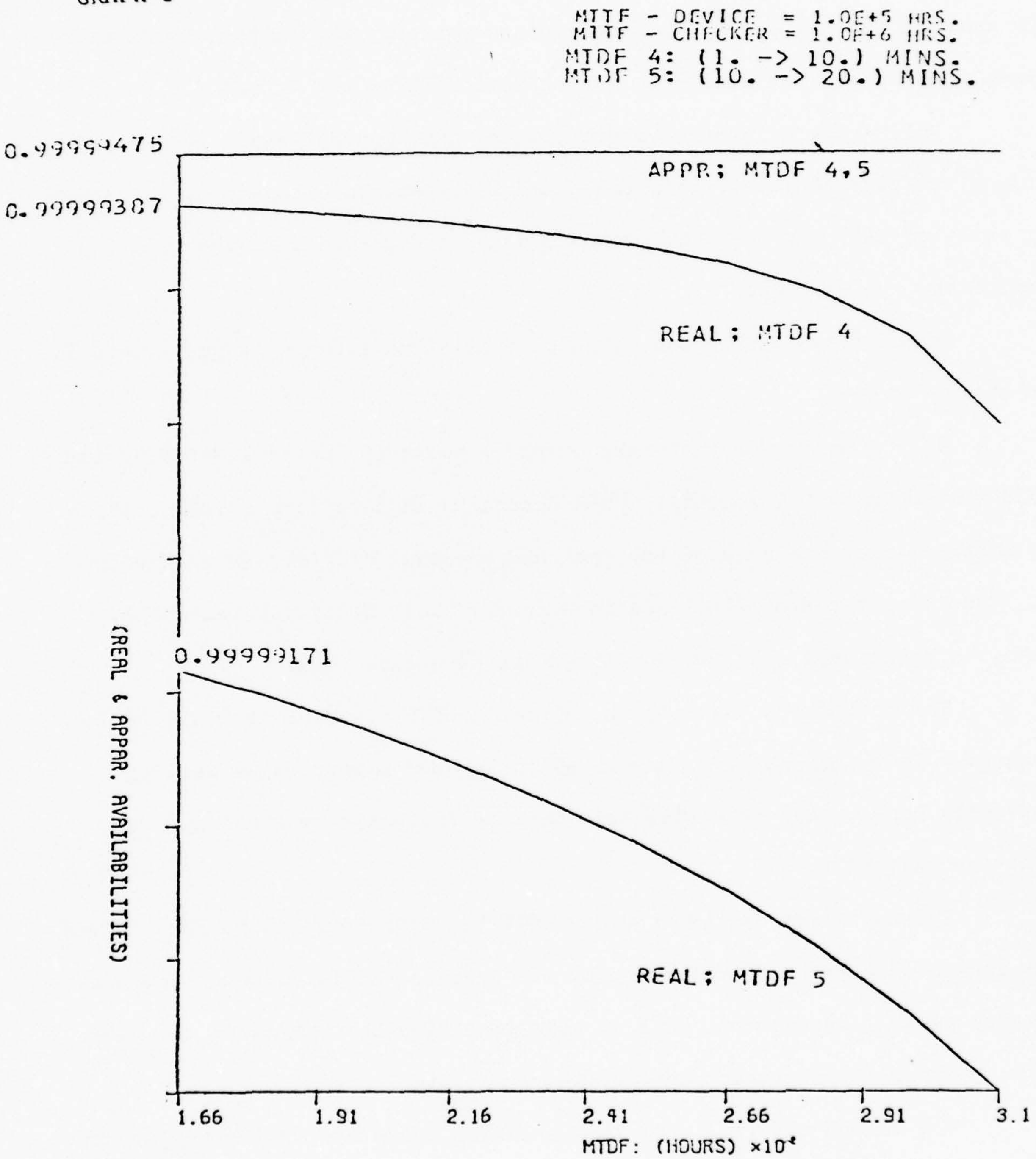
The bottom curve is the real availability for MTDF from 1.0 to 10. seconds. The increased latency is reflected by a greatly decreased real availability. Furthermore, the system's availability is now sensitive to small relative changes in the detection time. This is reflected by the steep descending slope of the real availability.

A similar pattern is seen for the remaining two MTDF time ranges in Graph 3. Again, the apparent availability shows no relative change. The real availabilities show sensitivity to latency increases.

An identical picture is produced when the MTTF's are decreased, and the MTTR's are increased (Cases II and III). The only difference for these cases of increased system complexity is that the absolute availability values are decreased.

Appendix I presents the graphs and tables for the remaining cases.

GRAPH 3


MTTF - DEVICE = 1.0E+5 HRS.
MTTF - CHECKER = 1.0E+6 HRS.
MTDF 4: (1. -> 10.) MINS.
MTDF 5: (10. -> 20.) MINS.
0.99999475
0.99999387
APPR; MTDF 4,5
REAL; MTDF 4
0.99999171
(REAL & APPAR. AVAILABILITIES)
REAL; MTDF 5
1.66
1.91
2.16
2.41
2.66
2.91
3.1
MTDF: (HOURS) ×10^{-2}

2. The Non-Maintained System

The same parameter values over the first three latency time ranges are used in Graph 4 for the non-maintained system. The bottom, horizontal curve, just above the abscissa, is the real MTTF of the system.

Unlike the maintained system, where the apparent availability was only a relative constant, this real MTTF is an absolute constant. As shown in equation (16) the real MTTF depends only on the MTTF's of the unit device and the checker.

Since these values are fixed over all five latency ranges, there is no change.

The almost horizontal line directly above is the apparent MTTF for MTDF from 0.01 to 0.1 seconds. When detection is done very quickly, there is little difference between the real and apparent MTTF's. (When detection is immediate, the two MTTF's are identical.) This first apparent MTTF, then, is relatively close to the immediate detection case.

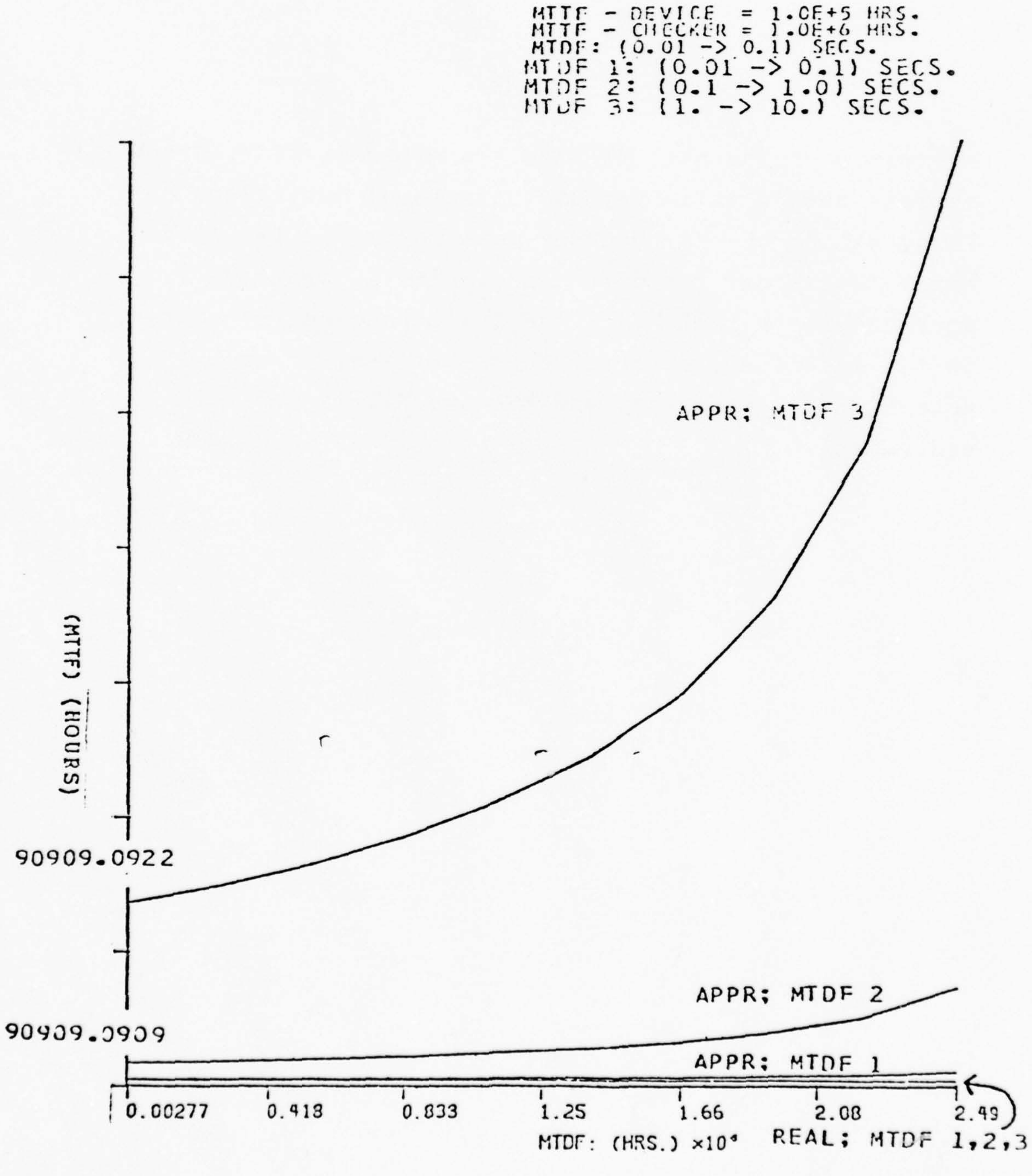
The next curve above is the apparent MTTF for MTDF from 0.1 to 1.0 seconds. This increased latency makes the system appear to be working correctly longer than it really is. This is reflected by the wider gap from the real MTTF.

The top curve is the apparent MTTF for MTDF from 1.0 to 10. seconds. The increased latency is reflected by the increased MTTF values. In this latency range, the apparent MTTF is more sensitive to changes in the MTDF. Thus, thesharply increased slope of the top curve.

The values and graphs of the remaining cases over all five latency ranges are given in Appendix II.

A clear picture of what is happening can be obtained by examining

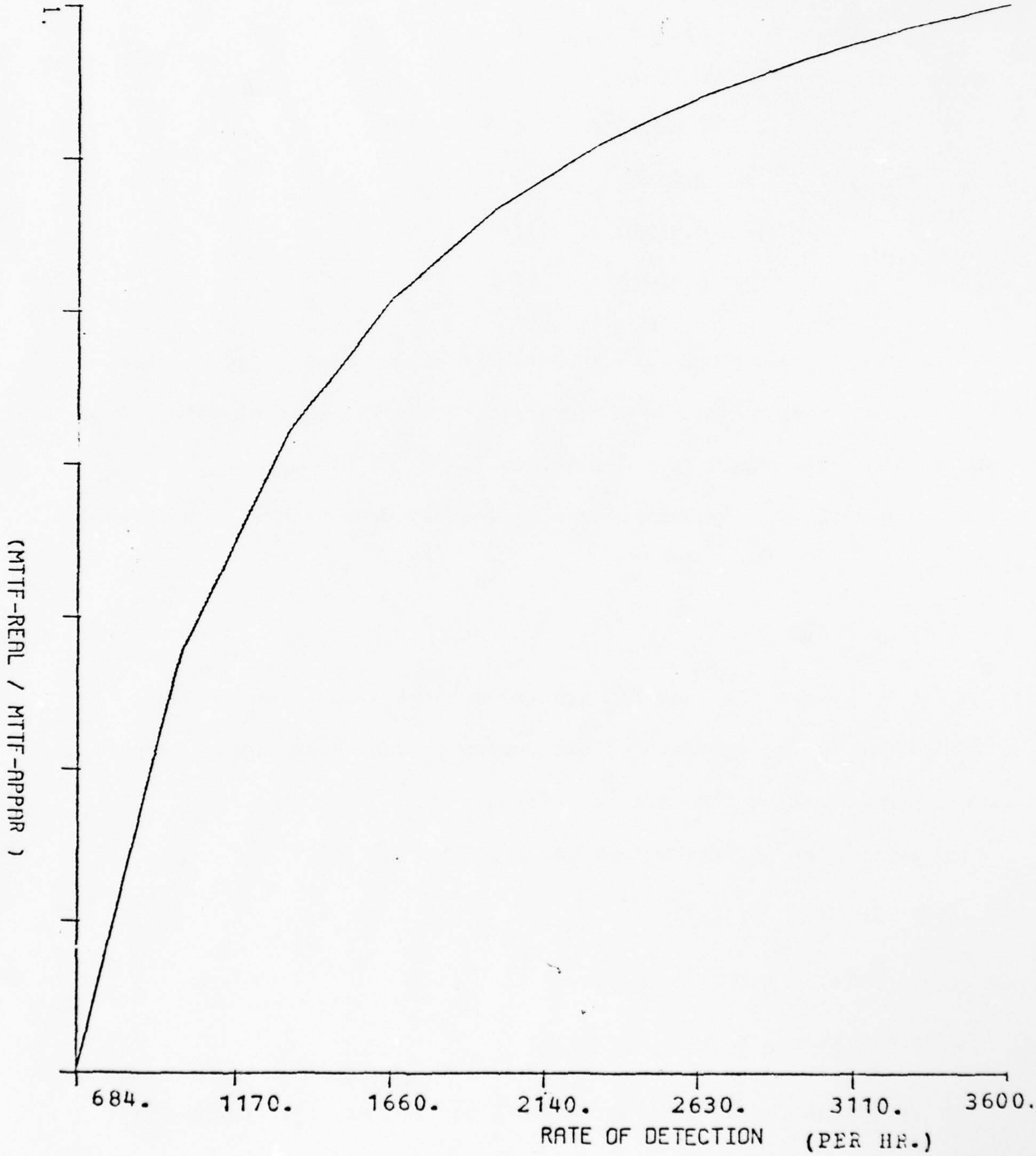
GRAPH 4


MTTF - DEVICE = 1.0E+5 HRS.
MTTF - CHECKER = 1.0E+6 HRS.
MTDF: (0.01 -> 0.1) SECS.
MTDF 1: (0.01 -> 0.1) SECS.
MTDF 2: (0.1 -> 1.0) SECS.
MTDF 3: (1. -> 10.) SECS.
APPR; MTDF 3
APPR; MTDF 2
APPR; MTDF 1
REAL; MTDF 1,2,3
(MTTF) (HOURS)
90909.0922
90909.0909
0.00277
0.418
0.833
1.25
1.66
2.08
2.49
MTDF: (HRS.) ×10⁴

the ratio of the real MTTF to the apparent MTTF. Graph 5 depicts such a ratio versus the rate of detection. Thus, as the checker becomes more powerful (ie. faster error detection) the more closely the apparent MTTF approaches the real MTTF. The limiting case, of course, is the infinitely powerful checker. Then, error detection is immediate, and the two MTTF's are equivalent.

GRAPH 5

DETECTOR EFFECTIVENESS



IV. ANALYSIS OF THE CHECKER

In a private communication, various detection rates ("δ") for corresponding values of checker moduli, M, were provided. [6]

$\delta(\text{secs}^{-1})$	M
$0.66\text{x}10^6$	3
$0.80\text{x}10^6$	5
$0.86\text{x}10^6$	7
$0.91\text{x}10^6$	11
$0.92\text{x}10^6$	13

For example, a system with a unit MTTF of 10^5 hours, and checker MTTF of 10^6 hours: if a modulus three addition checker were employed, then the rate of detection would be $0.66\text{x}10^6$ sec^{-1}.

It was desired to model these experimental data points by a curve of the form:

(18) $\delta = \delta_o M^a$

where "δ_o" and "a" are the constants to be determined.

In order to fit the data to the curve, it was necessary to transform this non-linear equation into a linear form.

By taking the logarithm (natural) of equation (18):

(19) $\ln(\delta) = \ln(\delta_o M^a)$

(20) $\ln(\delta) = \ln(M^a) + \ln(\delta_o)$

(21) $\ln(\delta) = a \ln(M) + \ln(\delta_o)$

By this means, a linear regression could be performed using the logarithms of "δ" and "M":

ln(δ)	ln(M)
13.40	1.10
13.59	1.61
13.66	1.95
13.72	2.40
13.73	2.56

A least squares regression on these data produced the following equation:

(22) $\delta = (0.22)M + 13.19$

Taking the anti-log of each side produces the equation of the desired form

(23) $\delta = (13.19)M^{0.22}$

To determine the goodness of the fit, the given experimental values for "δ" were compared to those values produced by equation (23).

M	δ-EXPERIMENTAL	δ-MODEL
3	0.66	0.69
5	0.80	0.77
7	0.86	0.83
11	0.91	0.92
13	0.92	0.95

A chi-squared test, with four degrees of freedom, yielded a value of 0.4622. Upon table look-up, this showed a level of confidence of 97.5%. Thus, the model provided an extremely good fit of the data points.

The larger the modulus "M" of the detector, the more powerful a checker it will be. It will also mean a more expensive checker. Thus, there is a trade off. It is desirable to have a powerful checker, capable of quickly detecting a large class of faults. It is not desirable, however, to have a needlessly large checker that incurs excessive cost.

In determining the size required of the checker, the importance of avoiding the undetected failure state must be considered. If it is extremely costly to the users to be in the failure state, then this implies that it is worth the extra cost of having a powerful detector. If the failure state can be tolerated for a longer period of time, then a smaller modulus checker can be used.

In general,

cost of checker = C_0 log(M)

K_F = a measure of the cost of being in the failure state.

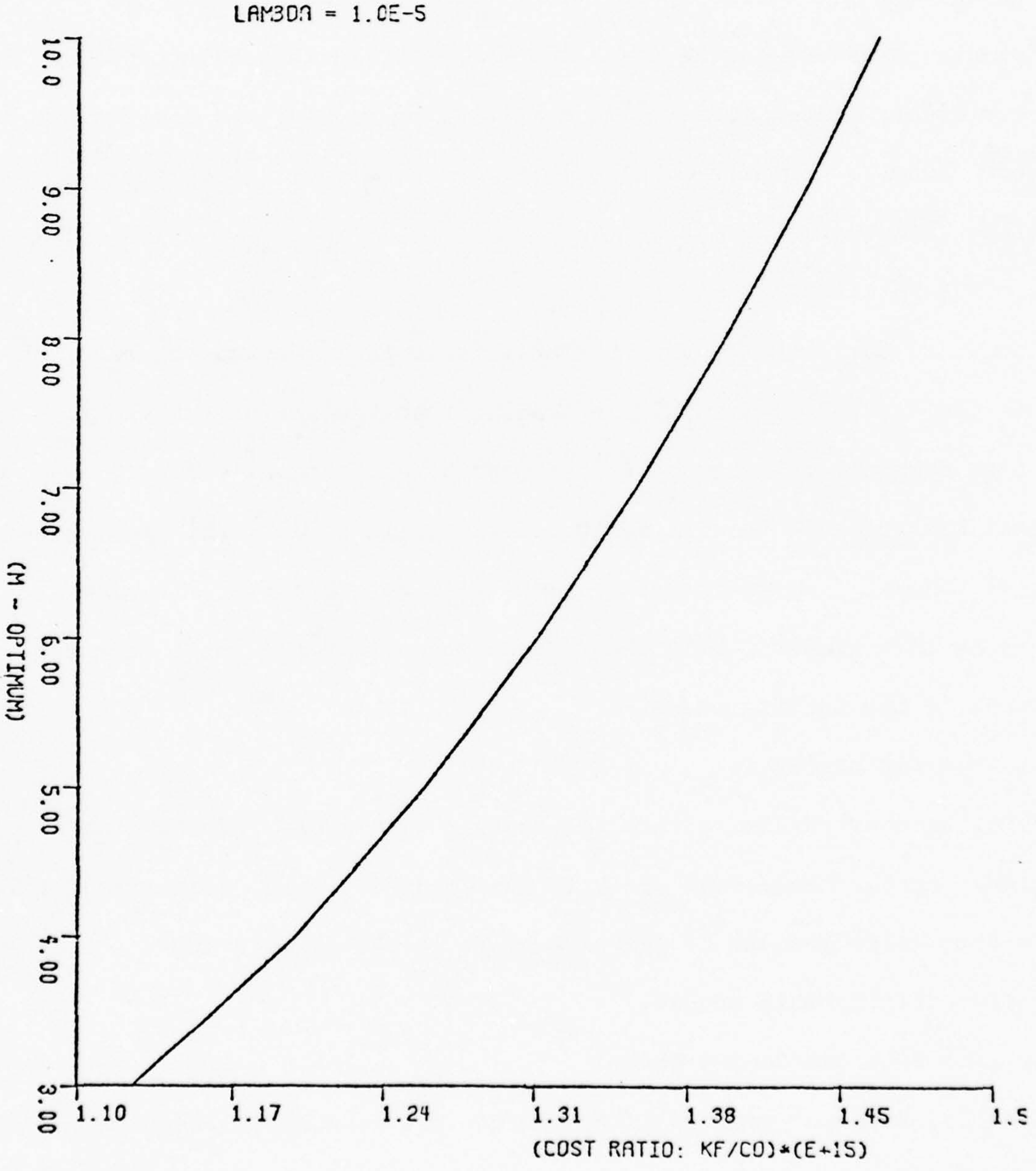
Then, the optimum modulus "M" can be related to these cost factors as follows: [5]

$$(24) \quad \text{M-opt.} = \left(\frac{K_F}{C_0} \cdot \frac{\lambda a}{\delta_0}\right)^{\frac{1}{2}}$$

where "a", and "δ_0" are from equations (18) and (23). This equation determines the optimum modulus for a given "cost ratio". (The cost ratio being K_F/C_0 - the cost of the failure state to the cost of the detector.)

The value of "M-opt" as a function of the cost ratio is shown in Graph 6. As expected, the larger the cost ratio (i.e., the more intolerable the failure state) the larger the required modulus of the detector.

GRAPH 6


LAMBDA = 1.0E-5
10.0
9.00
8.00
7.00
6.00
5.00
4.00
3.00
(M - OPTIMUM)
1.10
1.17
1.24
1.31
1.38
1.45
1.5
(COST RATIO: KF/CO)*(E+15)

V. SIMULATION OF THE MODELS

1. Looking for the Steady State

The first requirement of the simulation is to determine how long it takes to reach the steady state. The initial step is to generate random variates to determine the time to failure for the unit and the time to failure for the checker. Whichever is the smaller value determines which of these two events occurs first. The smaller time is kept and the larger one is thrown away. A counter is also incremented by one. This keeps account of how many cycles are executed.

a. The Working State

Consider, for example, that the unit fails first (which, on the average, is the case since its MTTF is smaller than that of the checker). Then two time accumulators must be incremented by this amount. First, a master clock has the time to unit failure added to it to mark the-progression of real time. Next, a special clock for the working state ("W") is also incremented by this amount. This timer keeps account of the total time the system spends in the working state.

b. The Failure State.

Next, another random variate is generated--this one for the time to detection. Again, the master clock is incremented. Then, another special clock, one that keeps account of the time spent in the failure state ("F") is also incremented by this amount.

c. The Detected-Repair State.

Finally, a random variate is generated that determines the time for repair. The master clock, as always, is incremented by this amount. A special clock that keeps account of time spent in the repair state ("D") is then incremented.

With repair time now accounted for, the system is back in the working state. The cycle is completed. The execution of such a sequence results in PRINT-OUT-1 being produced.

The cycle is repeated by again generating random variates for checker failure time and unit failure time, and incrementing the counter by one. If, for example, the checker fails first, a similar sequence occurs. First, the master clock and the working state clock are both incremented by this amount.

Next, a random variate is generated to give the time for checker repair. The master clock and the "C" state clock are both incremented. The cycle is completed and the system is once again back in the working state. The checker-failure-first sequence produces PRINT-OUT-2.

After ten executions of these cycles, a test is made to determine if the steady state has been reached. For the maintained system, this test is done in the following manner.

The real availability over the first five cycles is compared to the real availability over the first ten cycles. The apparent availabilities are similarly compared. If the (real/apparent) availability during the first five cycle executions is "reasonably" close to the (real/apparent) availability during the first ten cycles, then there has been little change in the system during this interval. Thus, the steady state has been reached. This would produce PRINT-OUT-3.

If either the real or apparent availabilities are not close over this time interval, then a message indicating this fact is printed, and program execution halts.

PRINT OUT 1

```
LOOKING FOR THE STEADY STATE
_COUNTER___________________1.000000000000000E+00______________________________________________

THE SYSTEM (UNIT) FAILS FIRST
*****************************

IN THE W - STATE  (WORKING)

CLOCK                          1.099491119384765E+05
CYCLE_TIME                     1.099491119384765E+05
SYS_FAILED_TIME                1.099491119384765E+05
    APPR_AVAIL_TIME            1.099491119384765E+05
    REAL_AVAIL_TIME            1.099491119384765E+05
    SYS_WORKING_STATE          1.099491119384765E+05
    SYS_WORKING_VAR            0.000000000000000E+00

IN THE F- STATE (FAILED)

CLOCK                          1.099491119412041E+05
CYCLE_TIME                     1.099491119412041E+05
SYS_DETECT_TIME                2.727606395880380E-06
    APPR_AVAIL_TIME            1.099491119412041E+05
    SYS_FAILURE_STATE          2.727606395880380E-06
    SYS_FAILURE_VAR            0.000000000000000E+00

IN THE D - STATE (DETECTED)

CLOCK                          1.099491124441029E+05
CYCLE_TIME                     1.099491124441029E+05
SYS_REPAIR_TIME                5.028988234698772E-04
    SYS_DETECT_STATE           5.028988234698772E-04
    SYS_DETECT_VAR             0.000000000000000E+00
```

PRINT OUT 2

```
LOOKING FOR THE STEADY STATE
COUNTER                        2.000000000000000E+00

THE CHECKER FAILS FIRST
******************************

IN THE W - STATE  (WORKING)

CLOCK                          1.150268842865826E+05
CYCLE_TIME                     5.077771842479705E+03
CHK_FAILED_TIME                5.077771842479705E+03
     APPR_AVAIL_TIME           5.077771842479705E+03
     REAL_AVAIL_TIME           5.077771842479705E+03
     SYS_WORKING_STATE         1.150268837809562E+05
     SYS_WORKING_VAR           9.999999908025374E-01

IN THE C - STATE (CHECKER REPAIR)

CLOCK                          1.150270605033638E+05
CYCLE_TIME                     5.077948059260845E+03
CHK_REPAIR_TIME                1.762167811393737E-01
     CHK_FAILURE_STATE         1.762167811393737E-01
     CHK_FAILURE_VAR           0.000000000000000E+00
```

PRINT OUT 3

```
LOCKING FOR THE STEADY STATE
THE STEADY STATE HAS BEEN REACHED
THE REAL AVAIL FOR THE FIRST 05 RUNS                9.999946763792459E-01
THE REAL AVAIL FOR THE FIRST 10 RUNS                9.999933105597668E-01
THE DIFFERENCE =          1.365819478987839E-06

THE APPR AVAIL FOR THE FIRST 05 RUNS                9.999946764184825E-C1
THE APPR AVAIL FOR THE FIRST 10 RUNS                9.999933105990664E-01
THE CIFFERENCE =          1.365819416287994E-C6
```

The manner in which the real and apparent availabilities are calculated is done as follows. After five cycles, the availabilities are determined by:

$$(25)\quad A_{REAL:5} = \frac{\text{WORKING STATE CLOCK}}{\text{MASTER CLOCK}}$$

$$(26)\quad A_{APPR:5} = \frac{(\text{WORKING STATE CLOCK}) \div (\text{FAILURE STATE CLOCK})}{\text{MASTER CLOCK}}$$

After ten cycles, $A_{REAL:10}$ and $A_{APPR:10}$ are calculated in the same manner. After ten cycles, all the clocks will have larger values than after five cycles, but it is hoped that the availabilities have remained fairly constant.

There is no steady state for the non-maintained system since the system goes down after each failure is detected. The equations of interest are the real and apparent MTTF.

The MTTF's are calculated as follows.

$$(27)\quad MTTF_R = \frac{\text{WORKING STATE CLOCK}}{\text{COUNTER}}$$

$$(28)\quad MTTF_A = \frac{(\text{WORKING STATE CLOCK}) + (\text{FAILURE STATE CLOCK})}{\text{COUNTER}}$$

2. Re-attaining the Steady State

All the clocks, counters, and various statistical accumulators are re-set to zero. The system then executes ten cycles in order to re-attain the steady state. During this period, as before, a print-out is generated for each cycle. Print-out-4 is an example.

PRINT OUT 4

```
RE-ATTAINING THE STEADY STATE
 COUNTER                          2.000000000000000E+C0

 THE SYSTEM (UNIT) FAILS FIRST
 *****************************

 IN THE W - STATE  (WORKING)

 CLOCK                            1.243592309213894E+05
 CYCLE_TIME                       1.894512176513671E+C4
 SYS_FAILED_TIME                  1.894512176513671E+C4
    APPR_AVAIL_TIME               1.894512176513671E+04
    REAL_AVAIL_TIME               1.894512176513671E+C4
    SYS_WORKING_STATE             1.243586540222167E+05
    SYS_WORKING_VAR               9.999890546417619E-01

 IN THE F- STATE (FAILED)

 CLOCK                            1.243592309219669E+05
 CYCLE_TIME                       1.894512176571421E+04
 SYS_DETECT_TIME                  5.774956610467698E-07
    APPR_AVAIL_TIME               1.894512176571421E+04
    SYS_FAILURE_STATE             1.175300445821549E-06
    SYS_FAILURE_VAR               3.216038859317083E-23

 IN THE D - STATE (DETECTED)

 CLOCK                            1.243593237960170E+C5
 CYCLE_TIME                       1.894521463976435E+04
 SYS_REPAIR_TIME                  9.287405014038085E-C2
    SYS_DETECT_STATE              6.697726249694824E-C1
    SYS_DETECT_VAR                2.995031857823057E-11
```

3. The Simulation

After ten executions of the cycle, the steady state is re-attained. The simulation consists of thirty additional cycle executions. All clocks and accumulators are kept accurate.

After thirty cycles, a listing is produced that shows the model's values of the availabilities or MTTF's according to equations (11), (12) or (16), (17), and simulated values using equations (25), (26) or (27), (28).

PRINT-OUT-6 is an example of the final statistics produced for the simulation of the maintained system. PRINT-OUT-7 is an example for the non-maintained system.

The complete program listing is given in Appendix III, including results from additional simulation runs.

4. Evaluation of the Simulation

As seen in PRINT-OUT-6, the model and the simulation availabilities agree to five decimal places. Furthermore, the variances for the simulation values are of the order of 10^{-5}. This means that the availability values are a very good indication of the state of the system during any particular time interval.

For the non-maintained system, the variances are close to the theoretic value of $(1/\lambda)^2$. Thus, the MTTF's give only a very general indication of how long the system is in the working state. There is a wide variation of values for any particular cycle.

5. Various Mechanisms of the Simulation

In presenting the simulation, a number of fine points were purposefully neglected in order that the main aspects of the program design would not be obscured.

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

PRINT OUT 5

```
FINAL STATISTICS

  SYSTEM PARAMETERS

     LAMBDA                            9.999999999999998E-06
     DELTA                             3.600000000000000E+05
     MU                                2.000000000000000E+00
     ALPHA                             9.999999999999999E-07
     BETA                              4.000000000000000E+00

  MTTF - DEVICE   = 1.0E+5 HRS.

  MTTF - CHECKER  = 1.0E+6 HRS.

  MTTR - DEVICE   = 0.5 HRS.

  MTTR - CHECKER  = 0.25 HRS.

  MTDF = 0.01 SECS.

  MODEL REAL AVAILABILITY                                    9.999947499997854E-01
  SIMULATION REAL AVAILABILITY                               9.999957244576326E-01
     DIFFERENCE BETWEEN MODEL & SIMULATE                     9.744578472192788E-07
     VARIANCE OF SIMULATION                                  6.308421610193649E-05

  MODEL APPR AVAILABILITY                                    9.999947500275630E-01
  SIMULATION APPR AVAILABILITY                               9.999957244786315E-01
     DIFFERENCE BETWEEN MODEL & SIMULATE                     9.744510684889240E-07
     VARIANCE OF SIMULATION                                  6.308398261376751E-05

  ALL AVERAGES ARE FOR AN AVERAGE CYCLE TIME

  AVG CYCLE TIME                                             9.816380073613468E+04

  PROB OF BEING IN SYS WORKING STATE                         9.999957244576326E-01
     AVG TIME IN SYS WORKING STATE                           9.816338103264570E+04
     VARIANCE OF SYS WORKING STATE                           6.308421610193649E-05

  PROB OF BEING IN SYS FAILURE STATE                         2.099891739138753E-11
     AVG TIME IN SYS FAILURE STATE                           2.061333542482718E-06
     VARIANCE OF SYS FAILURE STATE                           4.358046914919805E-12

  PROB OF BEING IN SYS DETECT STATE                          4.220498476207224E-06
     AVG TIME IN SYS DETECT STATE                            4.143001714255660E-01
     VARIANCE OF SYS DETECT STATE                            1.760457622184376E-01

  PROB OF BEING IN CHK FAILURE STATE                         5.502289303648897E-08
     AVG TIME IN CHK FAILURE STATE                           5.401256307959556E-03
     VARIANCE OF CHK FAILURE STATE                           2.992160928999514E-05
```

```
FINAL STATISTICS

SYSTEM PARAMETERS

  LAMBDA                      9.999999999999998E-06
  DELTA                       3.600000000000000E+05
  ALPHA                       9.999999999999999E-07

MTTF - DEVICE  = 1.0E+5 HRS.

MTTF - CHECKER = 1.0E+6 HRS.

MTDF = 0.01 SECS.


MODEL REAL MTTF               9.090909090909089E+04
SIMULATION REAL MTTF          8.465487725101411E+04
  DIFFERENCE BETWEEN MODEL & SIMULATE                         6.254213658076783E+03
  VARIANCE OF SIMULATION                                      7.163139130797053E+09


MODEL APPR MTTF               9.090909091161614E+04
SIMULATION APPR MTTF          8.465487725339923E+04
  DIFFERENCE BETWEEN MODEL & SIMULATE                         6.254213653216918E+03
  VARIANCE OF SIMULATION                                      7.163139130840251E+09


PROB OF BEING IN SYS WORKING STATE                            9.999999997182645E-01

PROB OF BEING IN SYS FAILURE STATE                            2.817461207529922E-11
```

PRINT OUT 6

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

One of these neglected areas concerns the method of generating the random variates that determine the times of failure, detection and, for the maintained case, repair. The first step is to generate a uniform random number between 0.0 and 1.0 This was done by means of a built-in procedure call available in PL/I. This procedure, "VARGEN", must be passed parameters that indicate the type of distribution, and the range of the random numbers desired [7].

Consider, for example, calculating the time to unit failure. The probability of such a failure in less than "t" time units is given by:

(29) $\Pr(T \leq t) = 1 - e^{-\lambda t}$

Then, the reliability, "R", of the unit:

(30) $R = e^{-\lambda t}$

(31) $\ln R = \ln(e^{-\lambda t})$

(32) $\ln R = -\lambda t$

(33) $t = -(1/\lambda)\ln(R)$

In order to calculate the time to detection or repair, the random number is factored by $-(1/\delta)$ or $-(1/\mu)$, respectively.

Another aspect that was glossed over concerned the test made to determine if the steady state had been reached. The strategy used here was that the means justify the ends.

Consider, for example, the maintained system. Assume that the original test, for the real availability was $|(A_{REAL:5} - A_{REAL:10})| < 10^{-5}$. For most runs, this test would not succeed. In fact, even 10^{-4} would often fail. So, a difference of 10^{-3} would be tried. If this worked, the final statistics would be checked to see if the simulation confirmed the model. If so, then this was an appropriate choice.

Finally, the number of cycles used in the simulation is an area previously not discussed. By the Law of Large Numbers [9] , a greater number of cycles, yielding similar results, will improve confidence in the simulation's findings. Thus, a simulation was conducted that ran for 400 cycles (100 to reattain the steady state, and 300 in the simulation.) See Print-out-7. The same parameters as in Print-out-5 (30 cycles) were used.

As seen from Print-out-8, the agreement between model and simulation is almost identical to the 30 cycle case. Thus, the strength of the simulation results is enhanced.

```
SIMULATION
COUNTER                   4.000000000000000E+02

THE SYSTEM (UNIT) FAILS FIRST
*****************************

IN THE W - STATE  (WORKING)

CLOCK                     3.789013505687338E+07
CYCLE_TIME                7.930074930191040E+04
SYS_FAILED_TIME           7.930074930191040E+04
   APPR_AVAIL_TIME        7.930074930191040E+04
   REAL_AVAIL_TIME        7.930074930191040E+04
   SYS_WORKING_STATE      3.788995937809813E+07
   SYS_WORKING_VAR        3.989770652064220E+02

IN THE F- STATE (FAILED)

CLOCK                     3.789013505687732E+07
CYCLE_TIME                7.930074930585479E+04
SYS_DETECT_TIME           3.944407569037542E-06
   APPR_AVAIL_TIME        7.930074930585479E+04
   SYS_FAILURE_STATE      9.890653950828308E-04
   SYS_FAILURE_VAR        4.079349284470406E-17

IN THE D - STATE (DETECTED)

CLOCK                     3.789013530402640E+07
CYCLE_TIME                7.930099645493483E+04
SYS_REPAIR_TIME           2.471490800380706E-01
   SYS_DETECT_STATE       1.678458220051834E+02
   SYS_DETECT_VAR         9.132781489821328E-06
```

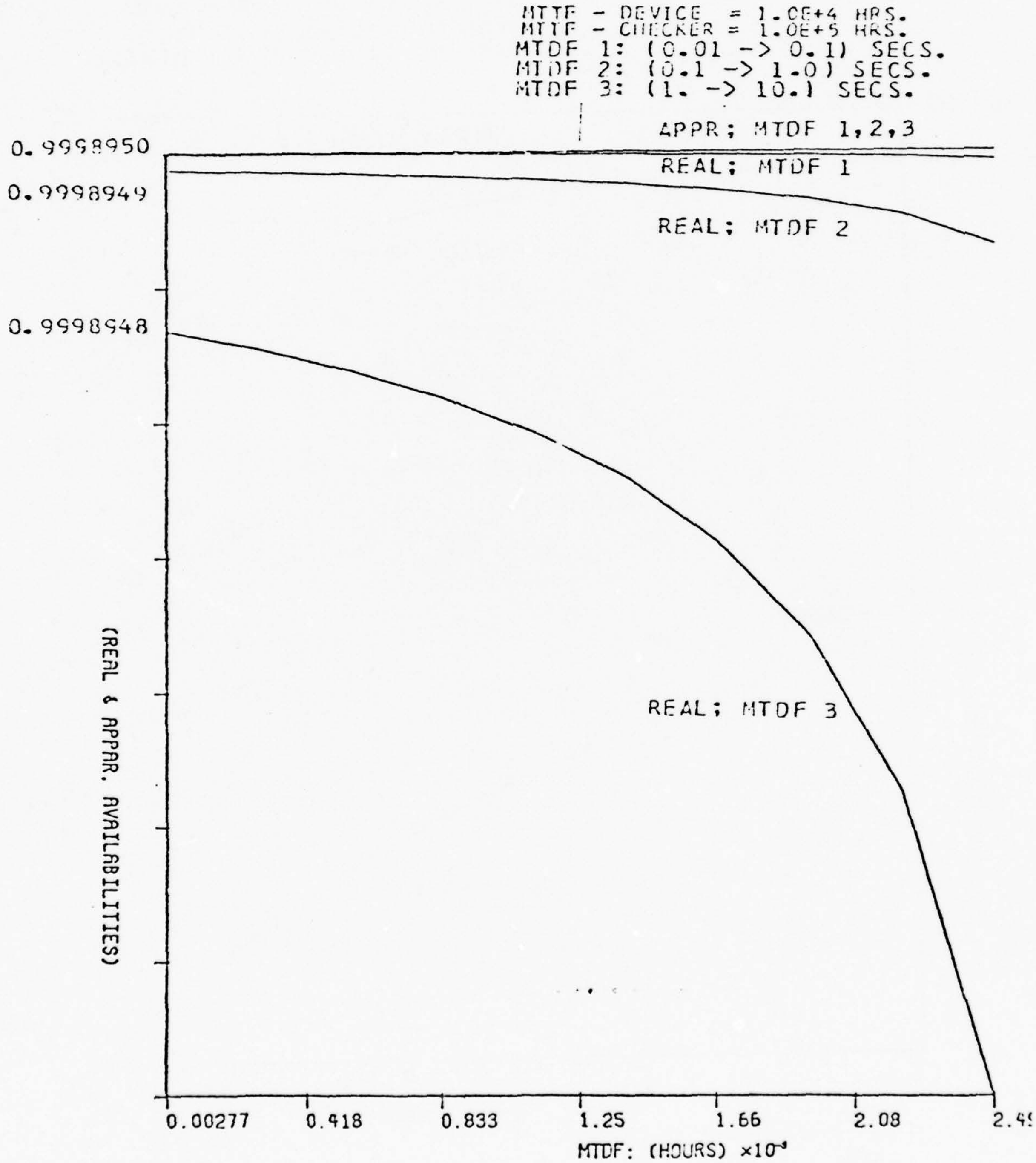
PRINT OUT 7

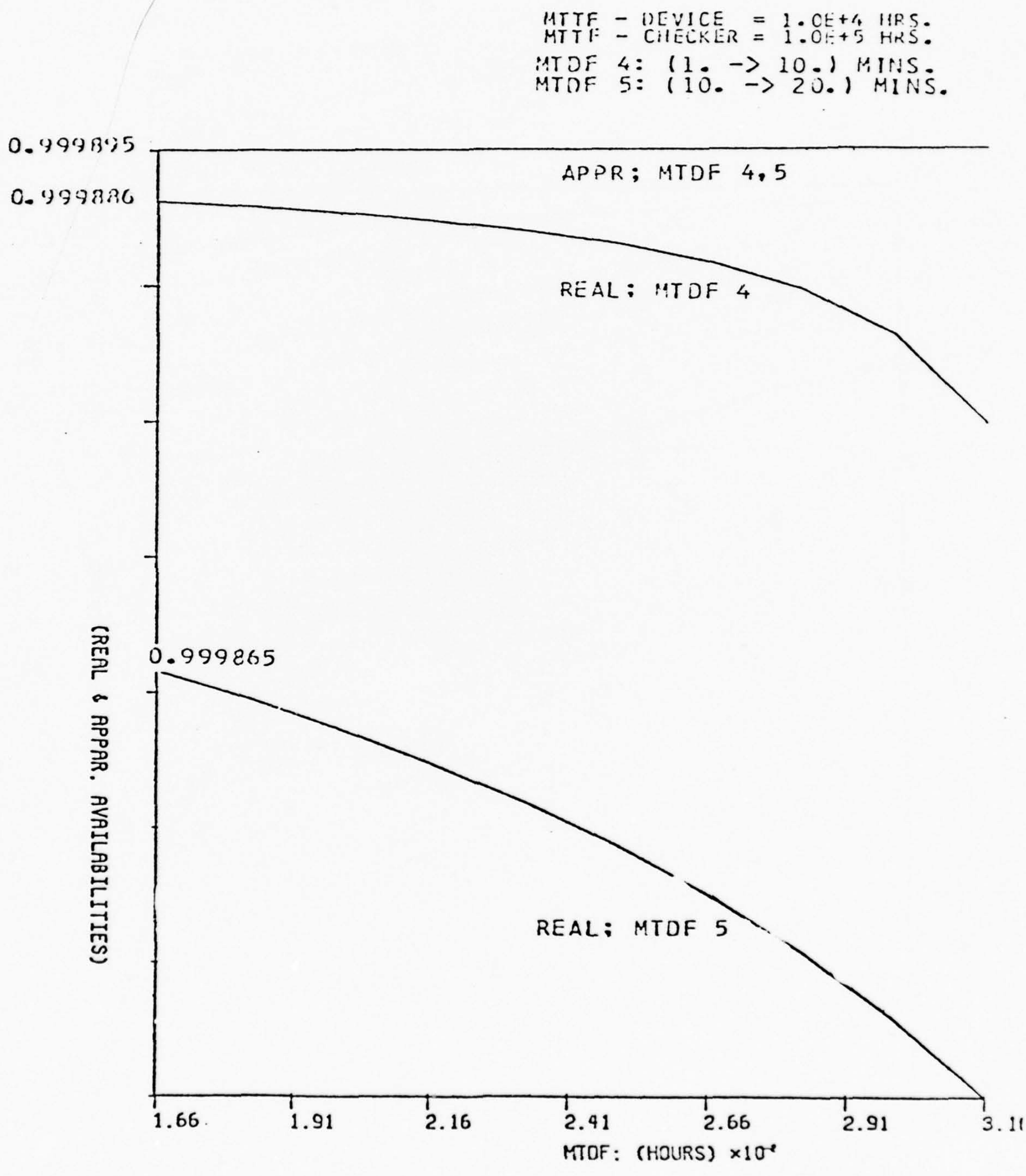
PRINT OUT 8

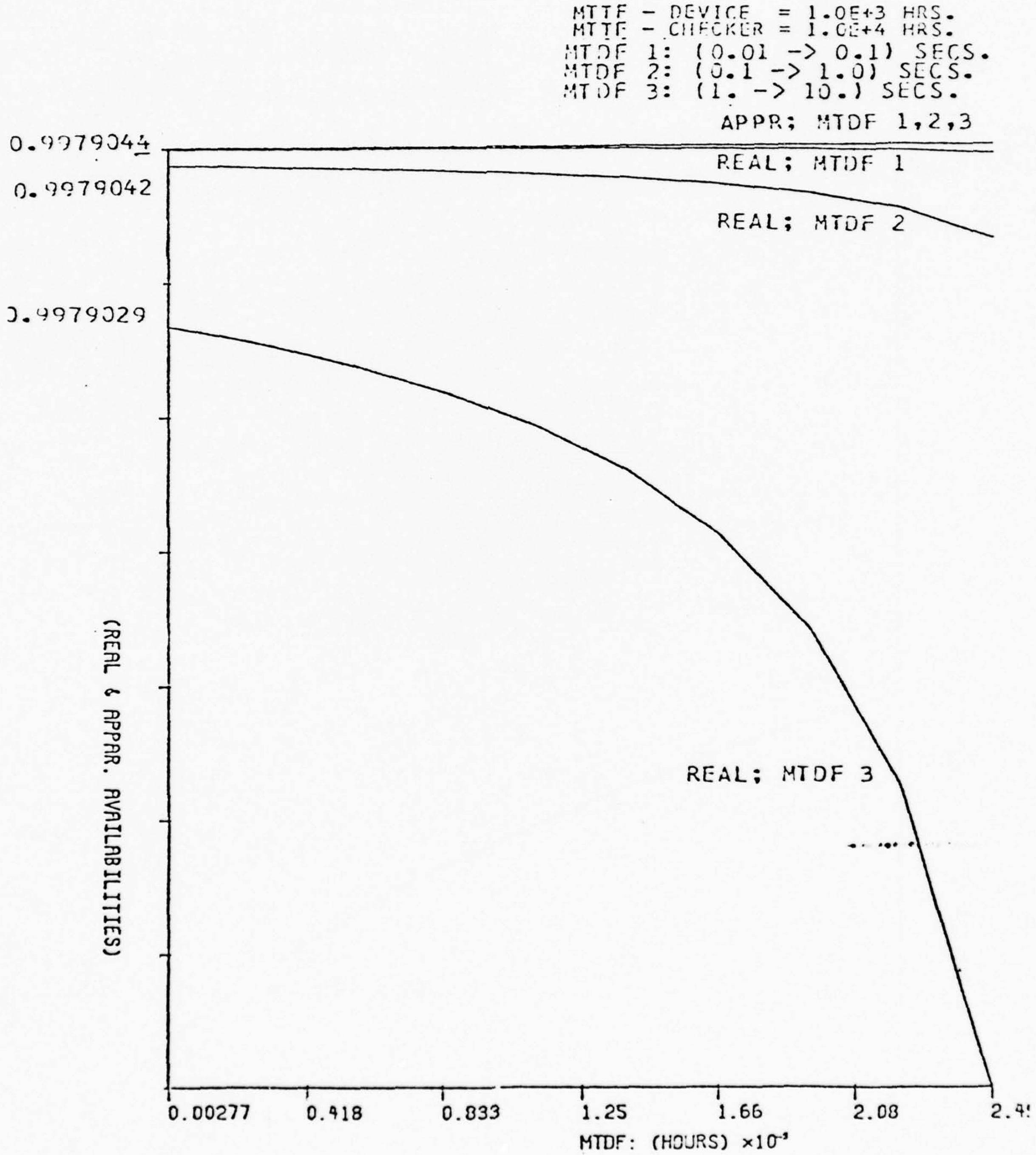
```
FINAL STATISTICS

SYSTEM PARAMETERS

   LAMBDA                           9.999999999999998E-06
   DELTA                            3.600000000000000E+05
   MU                               2.000000000000000E+00
   ALPHA                            9.999999999999999E-07
   BETA                             4.000000000000000E+00

MODEL REAL AVAILABILITY                                           9.999947499997854E-01
SIMULATICN REAL AVAILABILITY                                      9.999953569464227E-01
   DIFFERENCE BETWEEN MODEL & SIMULATE                            6.069466373104237E-07
   VARIANCE CF SIMULATION                                         4.818694891985028E-05

MODEL APPR AVAILABILITY                                           9.999947500275630E-01
SIMULATION APPR AVAILABILITY                                      9.999953569725262E-01
   DIFFERENCE BETWEEN MODEL & SIMULATE                            6.069449631634915E-07
   VARIANCE OF SIMULATION                                         4.818678508207066E-05

ALL AVERAGES ARE FOR AN AVERAGE CYCLE TIME

AVG CYCLE TIME                                                    9.472533826006599E+04

PROB OF BEING IN SYS WORKING STATE                                9.999953569464227E-01
   AVG TIME IN SYS WORKING STATE                                  9.472489844524534E+04
   VARIANCE OF SYS WORKING STATE                                  4.818694891985028E-05

PROB OF BEING IN SYS FAILURE STATE                                2.610350654983613E-11
   AVG TIME IN SYS FAILURE STATE                                  2.472663487707076E-06
   VARIANCE OF SYS FAILURE STATE                                  5.129308091679005E-12

PROB OF BEING IN SYS DETECT STATE                                 4.429802655979094E-06
   AVG TIME IN SYS DETECT STATE                                   4.196145550129585E-01
   VARIANCE OF SYS DETECT STATE                                   1.765176460618999E-01

PROB OF BEING IN CHK FAILURE STATE                                2.132248293276434E-07
   AVG TIME IN CHK FAILURE STATE                                  2.019779408350586E-02
   VARIANCE OF CHK FAILURE STATE                                  4.089732979601863E-04
```

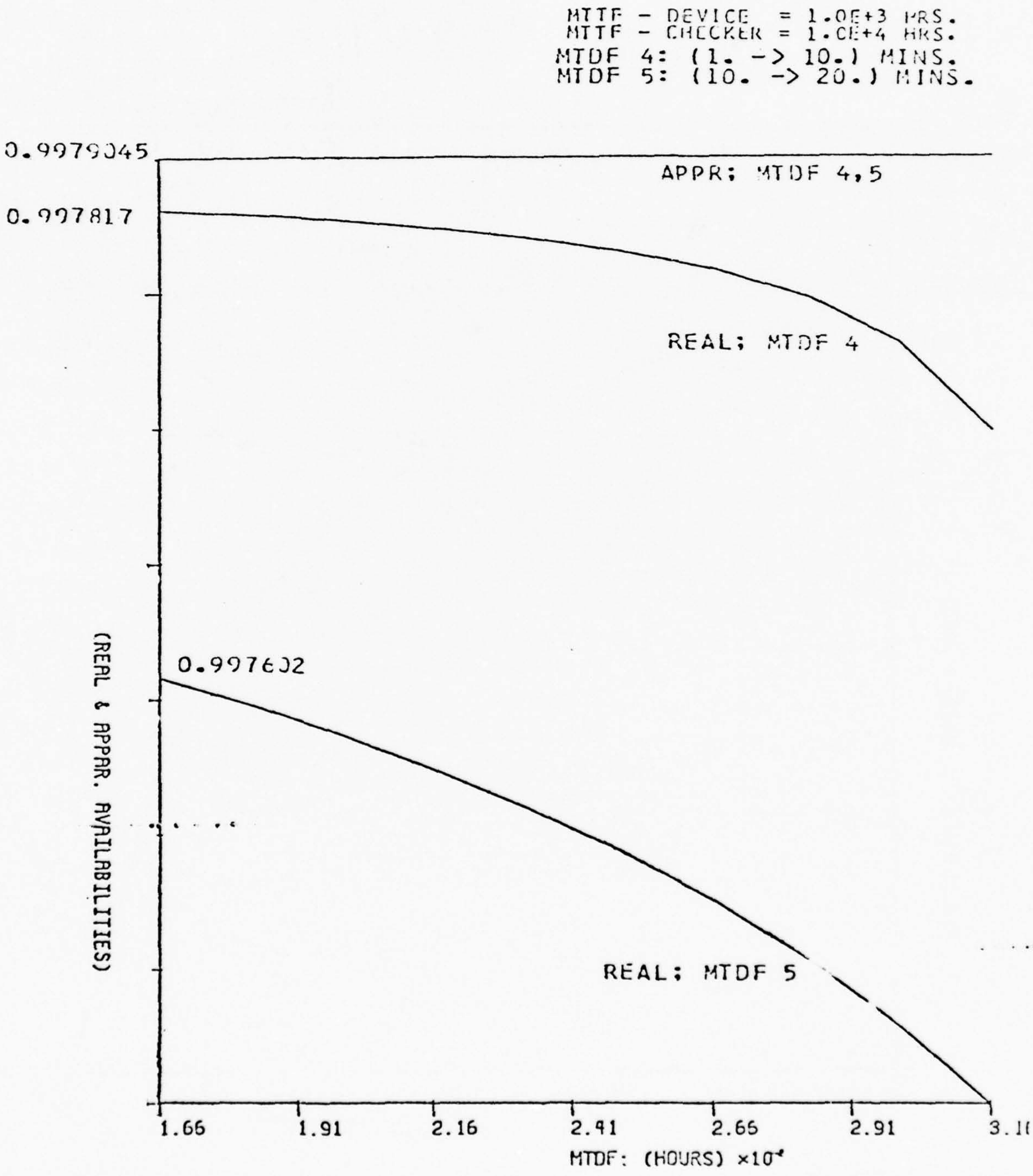
APPENDIX I

THE MAINTAINED SYSTEM


MTTF - DEVICE = 1.0E+4 HRS.
MTTF - CHECKER = 1.0E+5 HRS.
MTDF 1: (0.01 -> 0.1) SECS.
MTDF 2: (0.1 -> 1.0) SECS.
MTDF 3: (1. -> 10.) SECS.
APPR; MTDF 1,2,3
REAL; MTDF 1
REAL; MTDF 2
REAL; MTDF 3
0.9998950
0.9998949
0.9998948
(REAL & APPAR. AVAILABILITIES)
0.00277
0.418
0.833
1.25
1.66
2.08
2.4
MTDF: (HOURS) $\times 10^{-3}$


MTTF - DEVICE = 1.0E+4 HRS.
MTTF - CHECKER = 1.0E+5 HRS.
MTDF 4: (1. -> 10.) MINS.
MTDF 5: (10. -> 20.) MINS.
0.999895
0.999886
0.999865
APPR; MTDF 4,5
REAL; MTDF 4
REAL; MTDF 5
(REAL & APPAR. AVAILABILITIES)
1.66
1.91
2.16
2.41
2.66
2.91
3.1
MTDF: (HOURS) ×10⁻¹


MTTF - DEVICE = 1.0E+3 HRS.
MTTF - CHECKER = 1.0E+4 HRS.
MTDF 1: (0.01 -> 0.1) SECS.
MTDF 2: (0.1 -> 1.0) SECS.
MTDF 3: (1. -> 10.) SECS.
APPR; MTDF 1,2,3
REAL; MTDF 1
REAL; MTDF 2
REAL; MTDF 3
0.9979044
0.9979042
0.9979029
(REAL & APPAR. AVAILABILITIES)
0.00277
0.418
0.833
1.25
1.66
2.08
2.4
MTDF: (HOURS) ×10⁻³


MTTF - DEVICE = 1.0E+3 HRS.
MTTF - CHECKER = 1.0E+4 HRS.
MTDF 4: (1. -> 10.) MINS.
MTDF 5: (10. -> 20.) MINS.
0.9979045
0.997817
0.997602
APPR; MTDF 4,5
REAL; MTDF 4
REAL; MTDF 5
(REAL & APPAR. AVAILABILITIES)
1.66
1.91
2.16
2.41
2.66
2.91
3.1(
MTDF: (HOURS) $\times 10^{-2}$

```
MTTF - DEVICE  = 1.0E+5 HRS.
MTTF - CHECKER = 1.0E+6 HRS.
MTTR - DEVICE  = 0.5 HPS.
MTTR - CHECKER = 0.25 HRS.
MTDF:  0.1 SEC. -> 1.0 SEC.


MTDF                       REAL AVAIL                  APPR AVAIL
 2.777777777777777E-05      9.999947497497876E-01      9.999947500275640E-01
 3.052503052503052E-05      9.999947497223153E-01      9.999947500275640E-01
 3.387533875338753E-05      9.999947496888128E-01      9.999947500275643E-01
 3.805175038051750E-05      9.999947496470489E-01      9.999947500275644E-01
 4.340277777777777E-05      9.999947495935392E-01      9.999947500275648E-01
 5.050505050505050E-05      9.999947495225172E-01      9.999947500275651E-01
 6.038647342995168E-05      9.999947494237041E-01      9.999947500275657E-01
 7.507507507507507E-05      9.999947492768196E-01      9.999947500275665E-01
 9.920634920634920E-05      9.999947490355094E-01      9.999947500275678E-01
 1.461988304093567E-04      9.999947485655896E-01      9.999947500275702E-01
```

```
MTTF - DEVICE  = 1.0E+5 HRS.
MTTF - CHECKER = 1.0E+6 HRS.
MTTR - DEVICE  = 0.5 HRS.
MTTR - CHECKER = 0.25 HRS.
MTDF: 1.0 SEC. -> 10. SEC.

MTDF                     REAL AVAIL               APPR AVAIL
 2.777777777777777E-04    9.999947472498140E-01    9.999947500275772E-01
 3.052503052503052E-04    9.999947469750915E-01    9.999947500275785E-01
 3.387533875338753E-04    9.999947466400641E-01    9.999947500275803E-01
 3.805175038051750E-04    9.999947462224275E-01    9.999947500275825E-01
 4.340277777777777E-04    9.999947456873302E-01    9.999947500275852E-01
 5.050505050505050E-04    9.999947449771105E-01    9.999947500275891E-01
 6.038647342995168E-04    9.999947439889787E-01    9.999947500275942E-01
 7.507507507507507E-04    9.999947425201337E-01    9.999947500276017E-01
 9.920634920634920E-04    9.999947401070320E-01    9.999947500276147E-01
 1.461988304093567E-03    9.999947354078332E-01    9.999947500276393E-01
```

```
MTTF - DEVICE  = 1.0E+5 HRS.
MTTF - CHECKER = 1.0E+6 HRS.
MTTR - DEVICE  = 0.5 HRS.
MTTR - CHECKER = 0.25 HRS.
MTDF:  1.0 MIN. -> 10. MIN.

MTDF                    REAL AVAIL              APPR AVAIL
 1.666666666666666E-02   9.999945833626737E-01   9.999947500284376E-01
 1.831501831501831E-02   9.999945668793360E-01   9.999947500285240E-01
 2.032520325203252E-02   9.999945467777056E-01   9.999947500286296E-01
 2.283105022831050E-02   9.999945217195094E-01   9.999947500287609E-01
 2.604166666666667E-02   9.999944896136980E-01   9.999947500289297E-01
 3.030303030303031E-02   9.999944470005331E-01   9.999947500291534E-01
 3.623188405797102E-02   9.999943877126574E-01   9.999947500294646E-01
 4.504504504504507E-02   9.999942995820448E-01   9.999947500299273E-01
 5.952380952380958E-02   9.999941547960716E-01   9.999947500306875E-01
 8.771929824561415E-02   9.999938728445600E-01   9.999947500321678E-01
```

```
MTTF - DEVICE  = 1.0E+5 HRS.
MTTF - CHECKER = 1.0E+6 HRS.
MTTR - DEVICE  = 0.5 HRS.
MTTR - CHECKER = 0.25 HRS.
MTDF:  10. MIN. -> 20. MIN.


MTDF                        REAL AVAIL                APPR AVAIL
 1.666666666666666E-01   9.999930833811736E-01   9.999947500363124E-01
 1.754385964912280E-01   9.999929956630964E-01   9.999947500367729E-01
 1.851851851851851E-01   9.999928981985843E-C1   9.999947500372848E-01
 1.96078431372549CE-01   9.999927892676814E-01   9.999947500378564E-01
 2.083333333333333E-01   9.999926667204443E-C1   9.999947500384998E-01
 2.222222222222222E-01   9.999925278336116E-C1   9.999947500392291E-01
 2.380952380952381E-01   9.999923691058502E-01   9.999947500400624E-01
 2.564102564102564E-01   9.999921859584956E-C1   9.999947500410238E-01
 2.777777777777777E-01   9.999919722866672E-01   9.999947500421457E-01
 3.03C303030303030E-01   9.999917197655326E-01   9.999947500434713E-01
```

```
MTTF - DEVICE  = 1.0E+4 HRS.
MTTF - CHECKER = 1.0E+5 HRS.
MTTR - DEVICE  = 1.0 HRS.
MTTR - CHECKER = 0.5  HRS.
MTDF:  0.01 SEC. -> 0.1 SEC


MTDF                    REAL AVAIL              APPR AVAIL
 2.777777777777777E-06   9.998950107461233E-01   9.998950110238720E-01
 3.0525C3052503052E-06   9.998950107186564E-01   9.998950110238747E-01
 3.387533875338753E-C6   9.998950106851604E-01   9.998950110238782E-01
 3.805175038051750E-C6   9.998950106434051E-01   9.998950110238827E-01
 4.340277777777777E-06   9.998950105899060E-01   9.998950110238881E-01
 5.C5C505050505049E-06   9.998950105188982E-01   9.998950110238956E-01
 6.038647342995168E-06   9.998950104201048E-01   9.998950110239062E-01
 7.5075C7507507505E-06   9.998950102732496E-01   9.998950110239216E-01
 9.920634920634920E-06   9.998950100319875E-C1   9.998950110239468E-01
 1.461988304093567E-05   9.998950095621614E-01   9.998950110239963E-01
```

```
MTTF - DEVICE  = 1.0E+4 HRS.
MTTF - CHECKER = 1.0E+5 HRS.
MTTR - DEVICE  = 1.0 HRS.
MTTR - CHECKER = 0.5  HRS.
MTDF:  0.1 SEC. -> 1.C SEC.


MTDF                     REAL AVAIL               APPR AVAIL
 2.777777777777777E-05    9.998950082466483E-01    9.998950110241344E-01
 3.C52503052503052E-C5    9.998950079719806E-01    9.998950110241632E-01
 3.387533875338753E-05    9.998950076370200E-01    9.998950110241982E-01
 3.805175C38051750E-05    9.998950072194667E-01    9.998950110242422E-01
 4.340277777777777E-05    9.998950066844763E-01    9.998950110242982E-01
 5.C5C505C50505050E-05    9.998950059743980E-01    9.998950110243729E-01
 6.038647342995168E-05    9.998950049864633E-01    9.998950110244766E-01
 7.507507507507507E-05    9.998950035179115E-01    9.9989501102463C7E-01
 9.920634920634920E-05    9.998950011052908E-01    9.998950110248841E-01
 1.461988304093567E-04    9.998949964070296E-01    9.998950110253776E-01
```

```
MTTF - DEVICE  = 1.0E+4 HRS.
MTTF - CHECKER = 1.0E+5 HRS.
MTTR - DEVICE  = 1.0 HRS.
MTTR - CHECKER = 0.5  HRS.
MTDF: 1.0 SEC. -> 10. SEC.


MTDF                    REAL AVAIL              APPR AVAIL
 2.777777777777777E-04  9.998949832518980E-01   9.998950110267587E-01
 3.C52503052503052E-04  9.998949805052222E-01   9.998950110270470E-01
 3.387533875338753E-C4  9.998949771556179E-01   9.998950110273990E-01
 3.805175038051750E-04  9.998949729800835E-01   9.998950110278374E-01
 4.34C277777777777E-04  9.998949676301800E-01   9.998950110283990E-01
 5.050505050505050E-04  9.998949605293992E-01   9.998950110291446E-01
 6.C38647342995168E-04  9.998949506500520E-01   9.998950110301819E-01
 7.507507507507507E-04  9.998949359645364E-01   9.998950110317238E-01
 9.920634920634920E-04  9.998949118383332E-01   9.998950110342571E-01
 1.461988304093567E-03  9.998948648557305E-01   9.998950110391902E-01
```

```
MTTF - DEVICE  = 1.0E+4 HRS.
MTTF - CHECKER = 1.0E+5 HRS.
MTTR - CEVICE  = 1.0 HRS.
MTTR - CHECKER = 0.5  HRS.
MTDF:  1.0 MIN. -> 10. MIN.


MTDF                      REAL AVAIL              APPR AVAIL
 1.666666666666666E-02   9.998933447098979E-01   9.998950111988057E-01
 1.831501831501831E-02   9.998931799099195E-01   9.998950112161097E-01
 2.032520325203252E-02   9.998929789344092E-01   9.998950112372120E-01
 2.283105022831050E-02   9.998927284034072E-01   9.998950112635177E-01
 2.604166666666667E-02   9.998924074107444E-01   9.998950112972220E-01
 3.030303030303031E-02   9.998919813662555E-01   9.998950113419566E-01
 3.623188405797102E-02   9.998913886093099E-01   9.998950114041961E-01
 4.504504504504507E-02   9.998905074854193E-01   9.998950114967144E-01
 5.952380952380958E-02   9.998890599281128E-01   9.998950116487076E-01
 8.771929824561415E-02   9.998862410127550E-01   9.998950119446937E-01
```

```
MTTF - DEVICE   = 1.0E+4 HRS.
MTTF - CHECKER = 1.0E+5 HRS.
MTTR - DEVICE   = 1.0 HRS.
MTTR - CHECKER = 0.5  HRS.
MTDF:  10. MIN. -> 20. MIN.

MTDF                   REAL AVAIL             APPR AVAIL
 1.666666666666666E-01  9.998783481343105E-01  9.998950127734460E-01
 1.754385964912280E-01  9.998774711555086E-01  9.998950128655288E-01
 1.851851851851851E-01  9.998764967364219E-01  9.998950129678428E-01
 1.960784313725490E-01  9.998754076820428E-01  9.998950130821936E-01
 2.083333333333333E-01  9.998741824987025E-01  9.998950132108379E-01
 2.222222222222222E-01  9.998727939612129E-01  9.998950133566342E-01
 2.380952380952381E-01  9.998712070659472E-01  9.998950135232582E-01
 2.564102564102564E-01  9.998693760392073E-01  9.998950137155160E-01
 2.777777777777777E-01  9.998672398498201E-01  9.998950139398158E-01
 3.030303030303030E-01  9.998647152741305E-01  9.998950142048964E-01
```

```
MTTF - DEVICE   = 1.0E+3 HRS.
MTTF - CHECKER = 1.0E+4 HRS.
MTTR - DEVICE   = 2.0 HRS.
MTTR - CHECKER = 1.0  HRS.
MTDF:  0.01 SEC. -> 0.1 SEC


MTDF                       REAL AVAIL                 APPR AVAIL
  2.777777777777777E-06    9.979043979922596E-01     9.979044007642163E-01
  3.C52503C52503052E-C6    9.979043977186846E-01     9.979044007647907E-01
  3.387533875338753E-06    9.979043973850563E-01     9.979044007654913E-01
  3.805175038051750E-06    9.979043969691640E-01     9.979044007663648E-01
  4.340277777777777E-C6    9.979043964363014E-01     9.979044007674838E-01
  5.050505050505049E-06    9.979043957290478E-C1     9.979044007689691E-01
  6.038647342995168E-06    9.979043947450427E-01     9.979044007710355E-01
  7.507507507507505E-06    9.979043932823323E-C1     9.979044007741070E-01
  9.920634920634920E-06    9.979043908793083E-01     9.979044007791534E-01
  1.461988304093567E-05    9.979043861997352E-01     9.979044007889806E-01
```

```
MTTF - DEVICE  = 1.0E+3 HRS.
MTTF - CHECKER = 1.0E+4 HRS.
MTTR - DEVICE  = 2.0 HRS.
MTTR - CHECKER = 1.0  HRS.
MTDF:  0.1 SEC. -> 1.0 SEC.

MTDF                      REAL AVAIL               APPR AVAIL
 2.777777777777777E-05     9.979043730969305E-01    9.979044008164964E-01
 3.052503052503052E-05     9.979043703611801E-01    9.979044008222414E-01
 3.387533875338753E-05     9.979043670248993E-01    9.979044008292478E-01
 3.805175038051750E-05     9.979043628659737E-01    9.979044008379816E-01
 4.34C277777777777E-05     9.979043575373505E-01    9.979044008491716E-01
 5.C5C505050505050E-C5     9.979043504648144E-01    9.979044008640240E-01
 6.C38647342995168E-05     9.979043406247640E-01    9.979044008846880E-01
 7.507507507507507E-05     9.979043259976627E-01    9.979044009154048E-01
 9.920634920634920E-05     9.979043019674257E-C1    9.979044009658683E-01
 1.461988304093567E-04     9.979042551717043E-01    9.979044010641393E-01
```

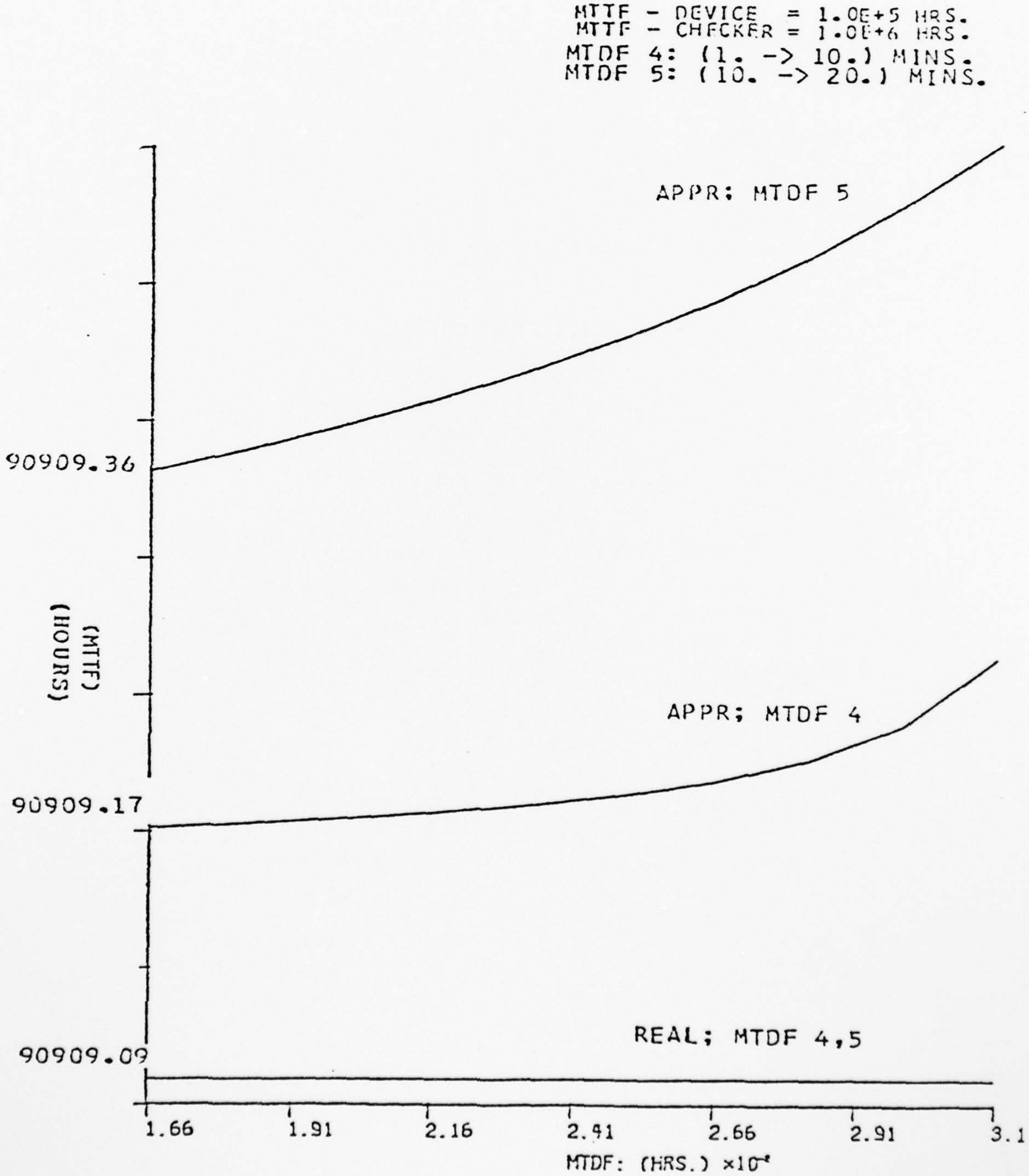
```
MTTF - DEVICE  = 1.0E+3 HRS.
MTTF - CHECKER = 1.0E+4 HRS.
MTTR - DEVICE  = 2.0 HRS.
MTTR - CHECKER = 1.0  HRS.
MTDF: 1.0 SEC. -> 10. SEC.

MTDF                   REAL AVAIL             APPR AVAIL
 2.777777777777777E-04  9.979041241437081E-01  9.979044013392982E-01
 3.052503052503052E-04  9.979040967862188E-01  9.979044013967491E-01
 3.387533875338753E-04  9.979040634234289E-01  9.979044014668108E-01
 3.805175038051750E-04  9.979040218342008E-01  9.979044015541481E-01
 4.340277777777777E-04  9.979039685480072E-01  9.979044016660491E-01
 5.050505050505050E-04  9.979038978227048E-01  9.979044018145723E-01
 6.038647342995168E-04  9.979037994223004E-01  9.979044020212131E-01
 7.507507507507507E-04  9.979036531514651E-01  9.979044023283819E-01
 9.920634920634920E-04  9.979034128494715E-01  9.979044028330160E-01
 1.461988304093567E-03  9.979029448932900E-01  9.979044038157241E-01
```

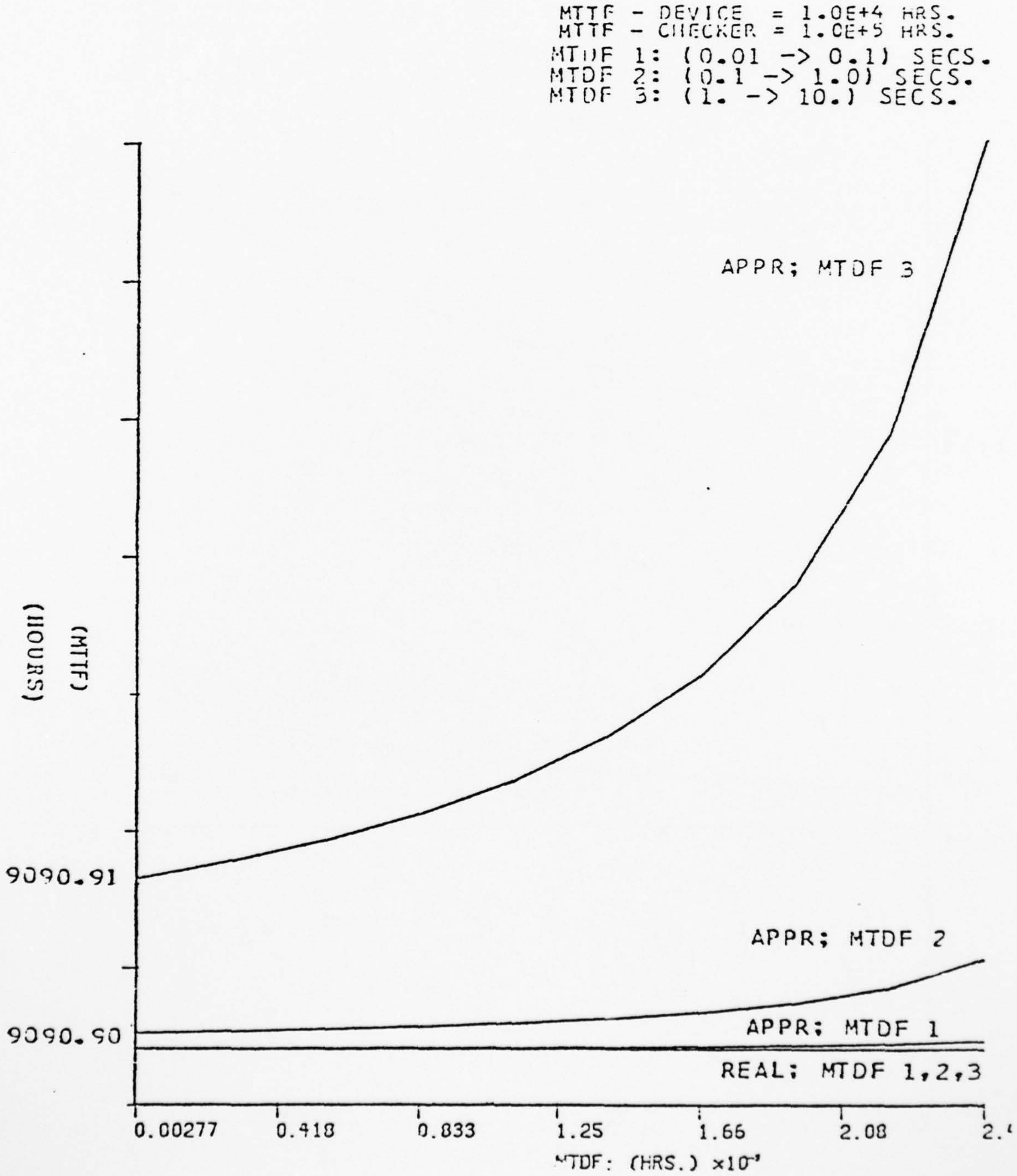
```
MTTF - DEVICE  = 1.0E+3 HRS.
MTTF - CHECKER = 1.0E+4 HRS.
MTTR - DEVICE  = 2.0 HRS.
MTTR - CHECKER = 1.0  HRS.
MTDF:  1.0 MIN. -> 10. MIN.

MTDF                   REAL AVAIL              APPR AVAIL
 1.666666666666666E-02  9.978878041478870E-01  9.979044356112894E-01
 1.831501831501831E-02  9.978861627548676E-01  9.979044390582148E-01
 2.032520325203252E-02  9.978841610633712E-01  9.979044432617668E-01
 2.283105022831050E-02  9.978816658153537E-01  9.979044485017878E-01
 2.604166666666667E-02  9.978784687970678E-01  9.979044552155260E-01
 3.030303030303031E-02  9.978742255135273E-01  9.979044641264217E-01
 3.623188405797102E-02  9.978683218747198E-01  9.979044765240630E-01
 4.504504504504507E-02  9.978595463245057E-01  9.979044949527185E-01
 5.952380952380958E-02  9.978451296842294E-01  9.979045252276630E-01
 8.771929824561415E-02  9.978170563696614E-01  9.979045841816236E-01
```

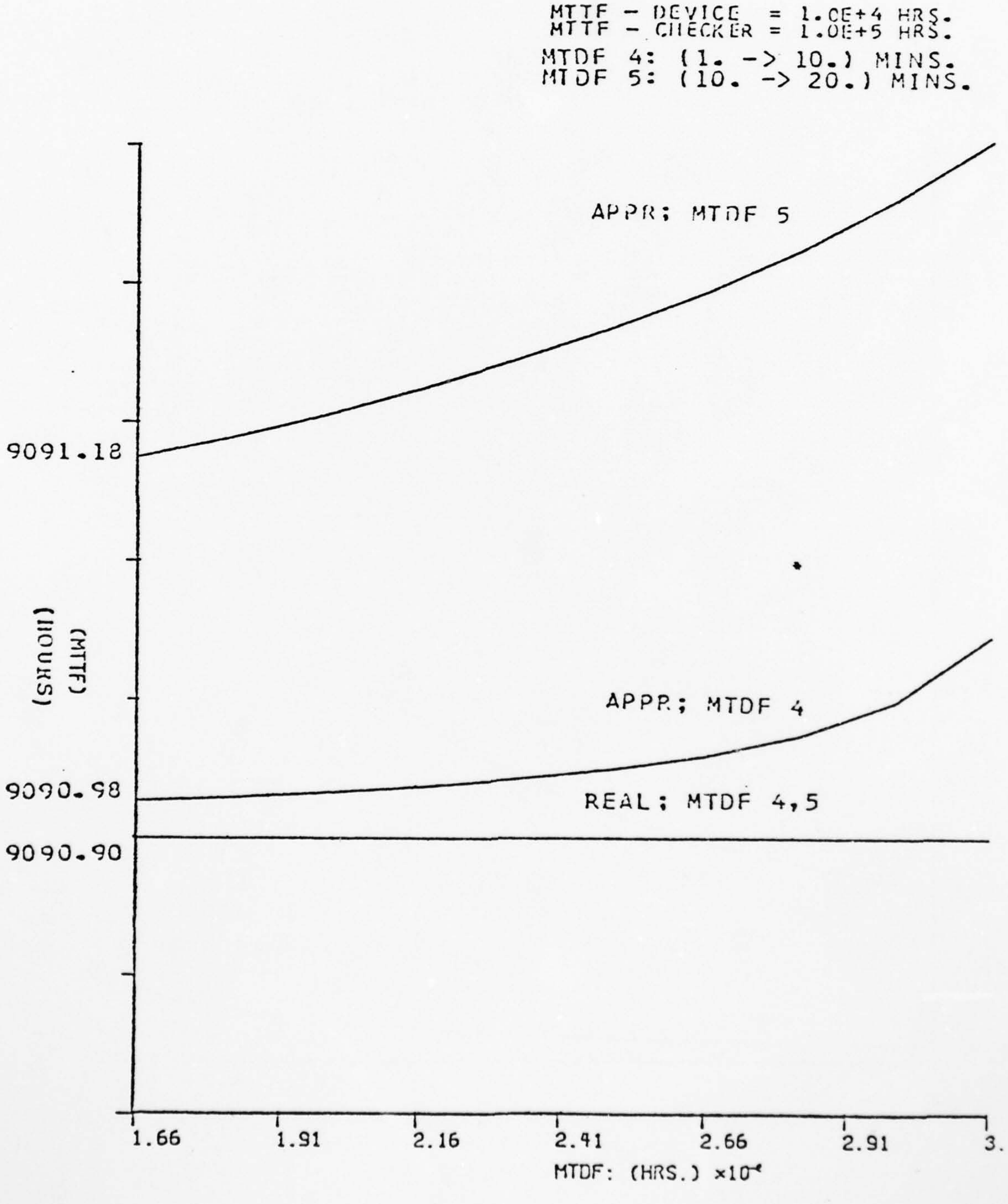
```
MTTF - DEVICE  = 1.0E+3 HRS.
MTTF - CHECKER = 1.0E+4 HRS.
MTTR - DEVICE  = 2.0 HRS.
MTTR - CHECKER = 1.0  HRS.
MTDF:  10. MIN. -> 20. MIN.

MTDF                    REAL AVAIL              APPR AVAIL
 1.666666666666666E-01   9.977384594918185E-01   9.979047492350670E-01
 1.754385964912280E-01   9.977297272697038E-01   9.979047675727336E-01
 1.851851851851851E-01   9.977200249799532E-01   9.979047879475420E-01
 1.960784313725490E-01   9.977091814676499E-01   9.979048107189180E-01
 2.083333333333333E-01   9.976969827980412E-01   9.979048363361241E-01
 2.222222222222222E-01   9.976831579997561E-01   9.979048653682004E-01
 2.380952380952381E-01   9.976673586994217E-01   9.979048985467312E-01
 2.564102564102564E-01   9.976491293592790E-01   9.979049368283454E-01
 2.777777777777777E-01   9.976278626377281E-01   9.979049814884608E-01
 3.030303030303030E-01   9.976027304084427E-01   9.979050342661422E-01
```

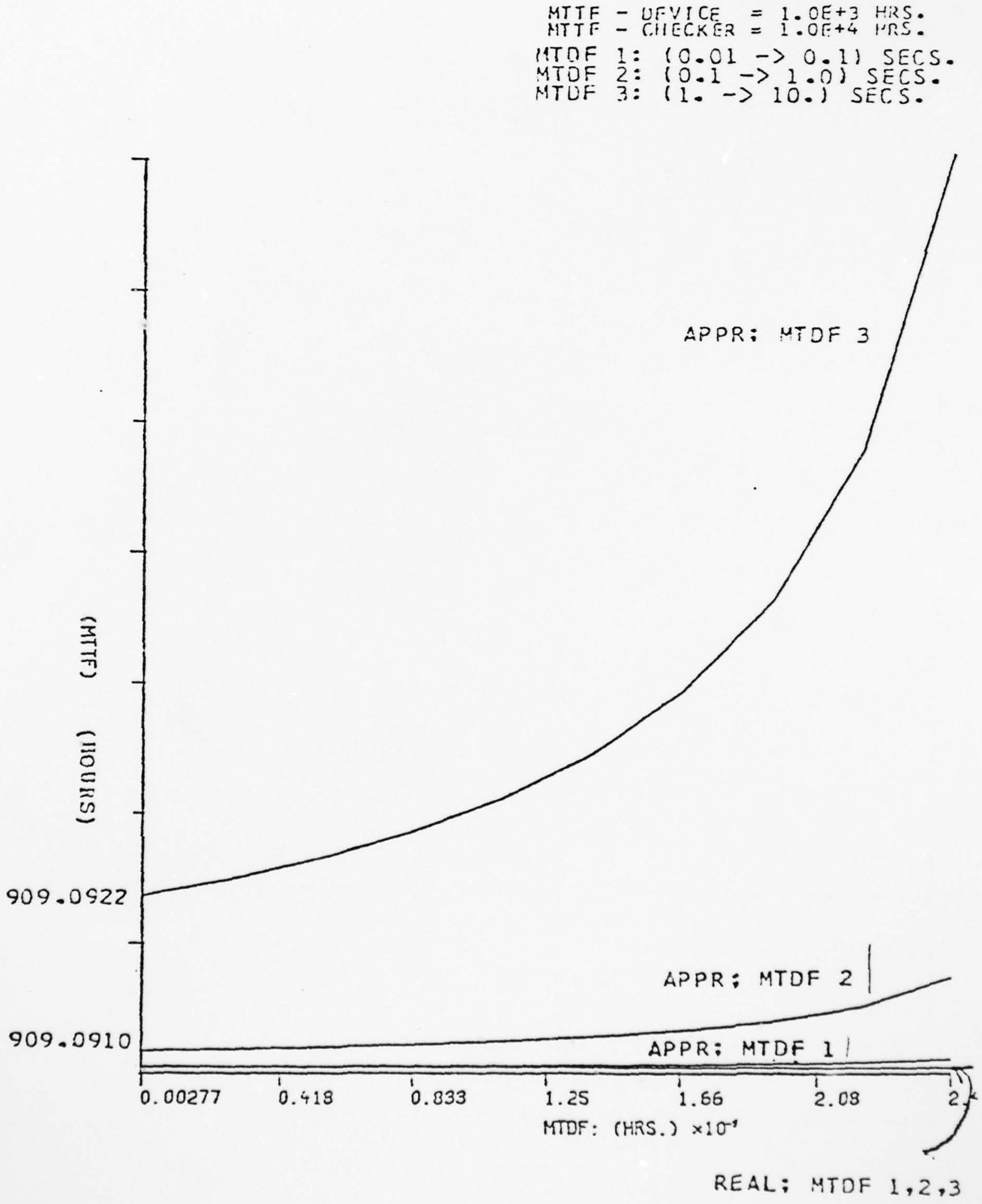
APPENDIX II

THE NON-MAINTAINED SYSTEM


MTTF - DEVICE = 1.0E+5 HRS.
MTTF - CHECKER = 1.0E+6 HRS.
MTDF 4: (1. -> 10.) MINS.
MTDF 5: (10. -> 20.) MINS.
APPR: MTDF 5
APPR; MTDF 4
REAL; MTDF 4,5
90909.36
90909.17
90909.09
(MTTF)
(HOURS)
1.66
1.91
2.16
2.41
2.66
2.91
3.1
MTDF: (HRS.) ×10⁻¹

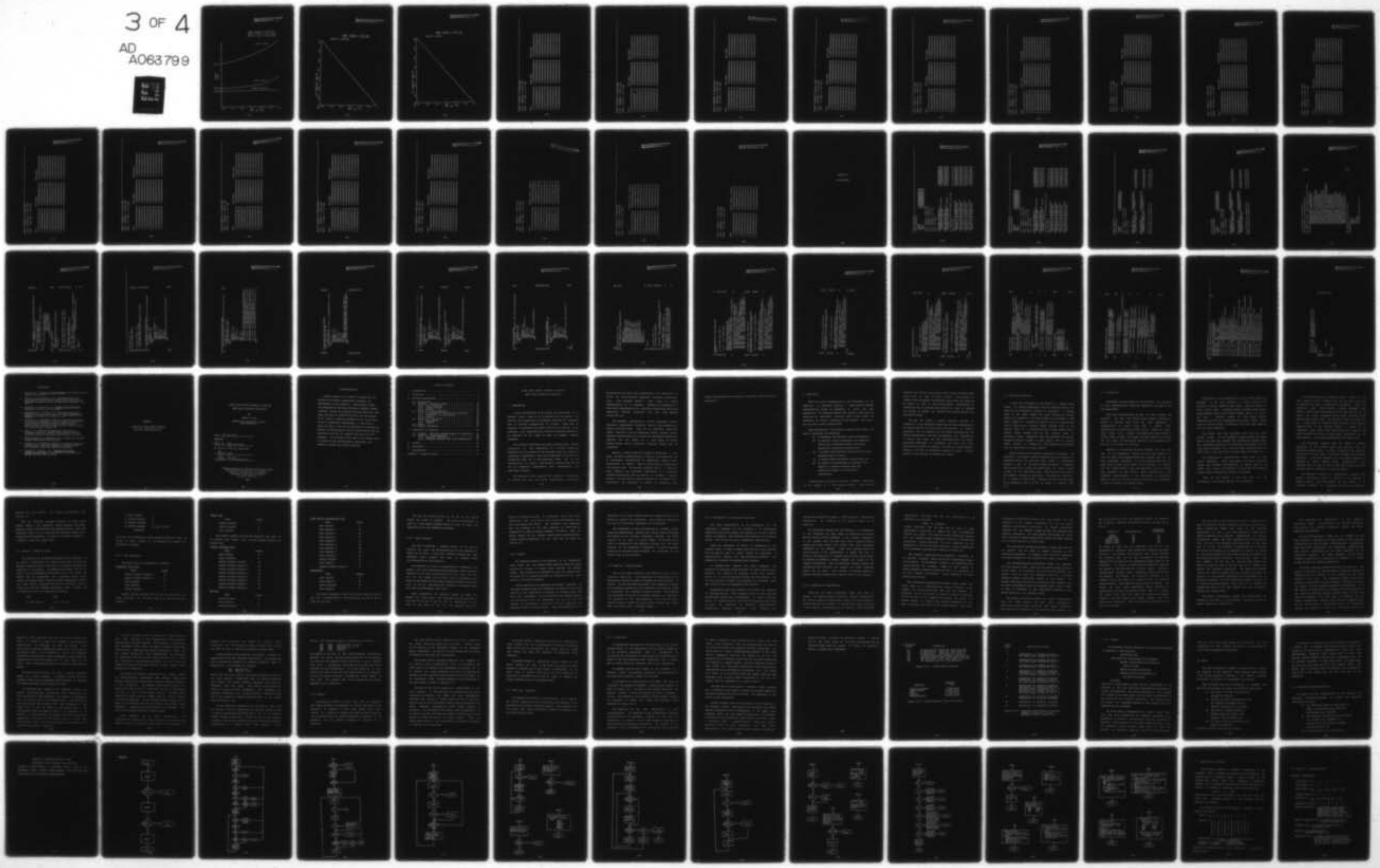

MTTF - DEVICE = 1.0E+4 HRS.
MTTF - CHECKER = 1.0E+5 HRS.
MTDF 1: (0.01 -> 0.1) SECS.
MTDF 2: (0.1 -> 1.0) SECS.
MTDF 3: (1. -> 10.) SECS.
APPR; MTDF 3
APPR; MTDF 2
APPR; MTDF 1
REAL; MTDF 1,2,3
(MTTF)
(HOURS)
9090.91
9090.90
0.00277
0.418
0.833
1.25
1.66
2.08
2.
MTDF: (HRS.) ×10-3

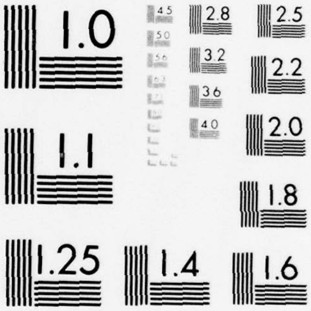

MTTF - DEVICE = 1.0E+4 HRS.
MTTF - CHECKER = 1.0E+5 HRS.
MTDF 4: (1. -> 10.) MINS.
MTDF 5: (10. -> 20.) MINS.
APPR; MTDF 5
APPR; MTDF 4
REAL; MTDF 4,5
9091.18
9090.98
9090.90
(MTTF)
(HOURS)
1.66
1.91
2.16
2.41
2.66
2.91
3.
MTDF: (HRS.) ×10⁻⁴


MTTF - DEVICE = 1.0E+3 HRS.
MTTF - CHECKER = 1.0E+4 HRS.
MTDF 1: (0.01 -> 0.1) SECS.
MTDF 2: (0.1 -> 1.0) SECS.
MTDF 3: (1. -> 10.) SECS.
APPR; MTDF 3
APPR; MTDF 2
APPR; MTDF 1
REAL; MTDF 1,2,3
(HOURS) (MTTF)
909.0922
909.0910
0.00277
0.418
0.833
1.25
1.66
2.08
2.
MTDF: (HRS.) ×10⁻¹

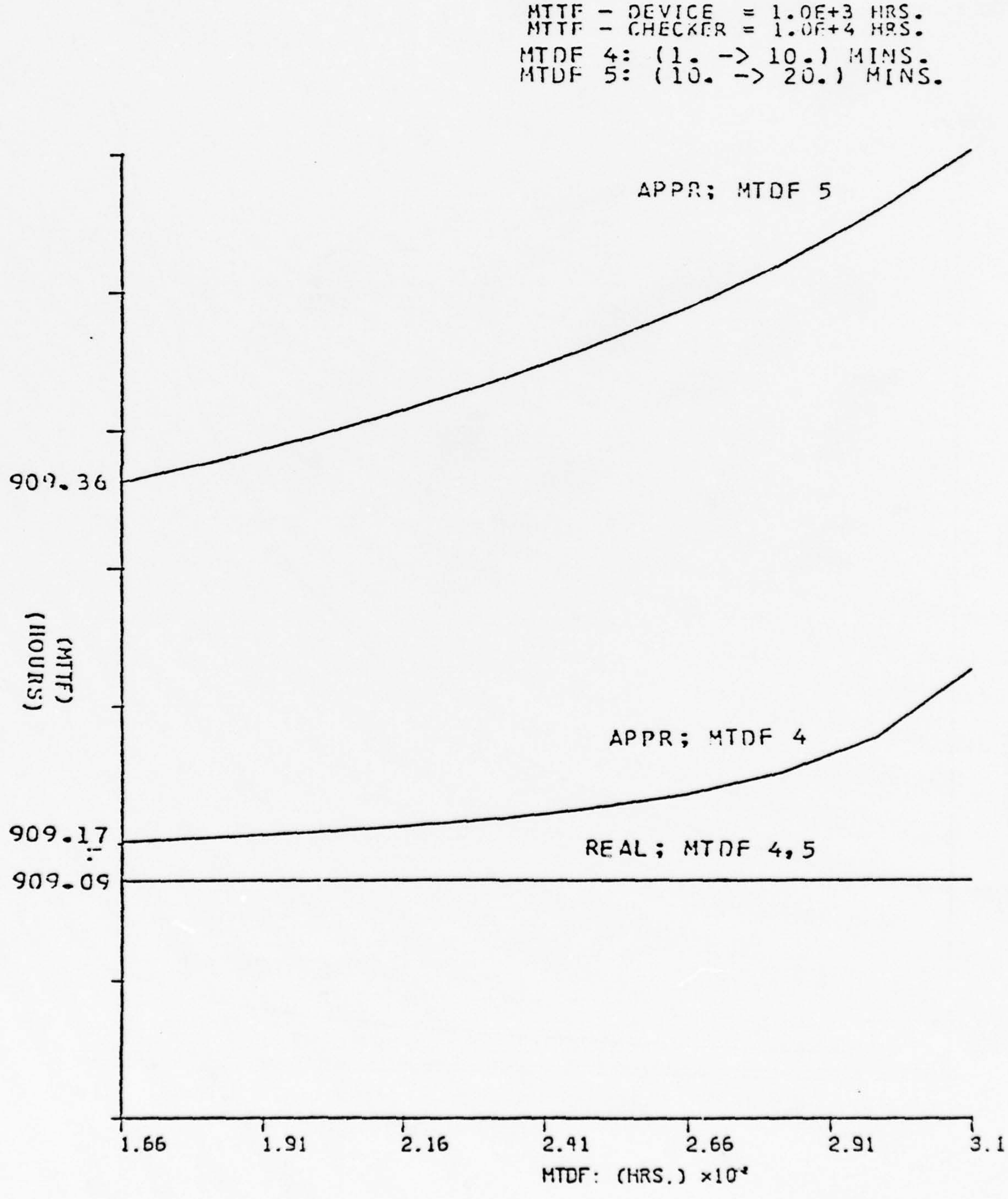
ID-A063 799 RESEARCH TRIANGLE INST RESEARCH TRIANGLE PARK NC SYST--ETC F/G 9/2
BASIC RESEARCH IN SUPPORT OF CONCURRENT FAULT MONITORING IN MOD--ETC(U)
JUN 78 J W GAULT, P N MARINOS, K S TRIVEDI N00039-77-C-0363
INCLASSIFIED RTI/1504/00-01-1 NL

3 OF 4
AD
A063799

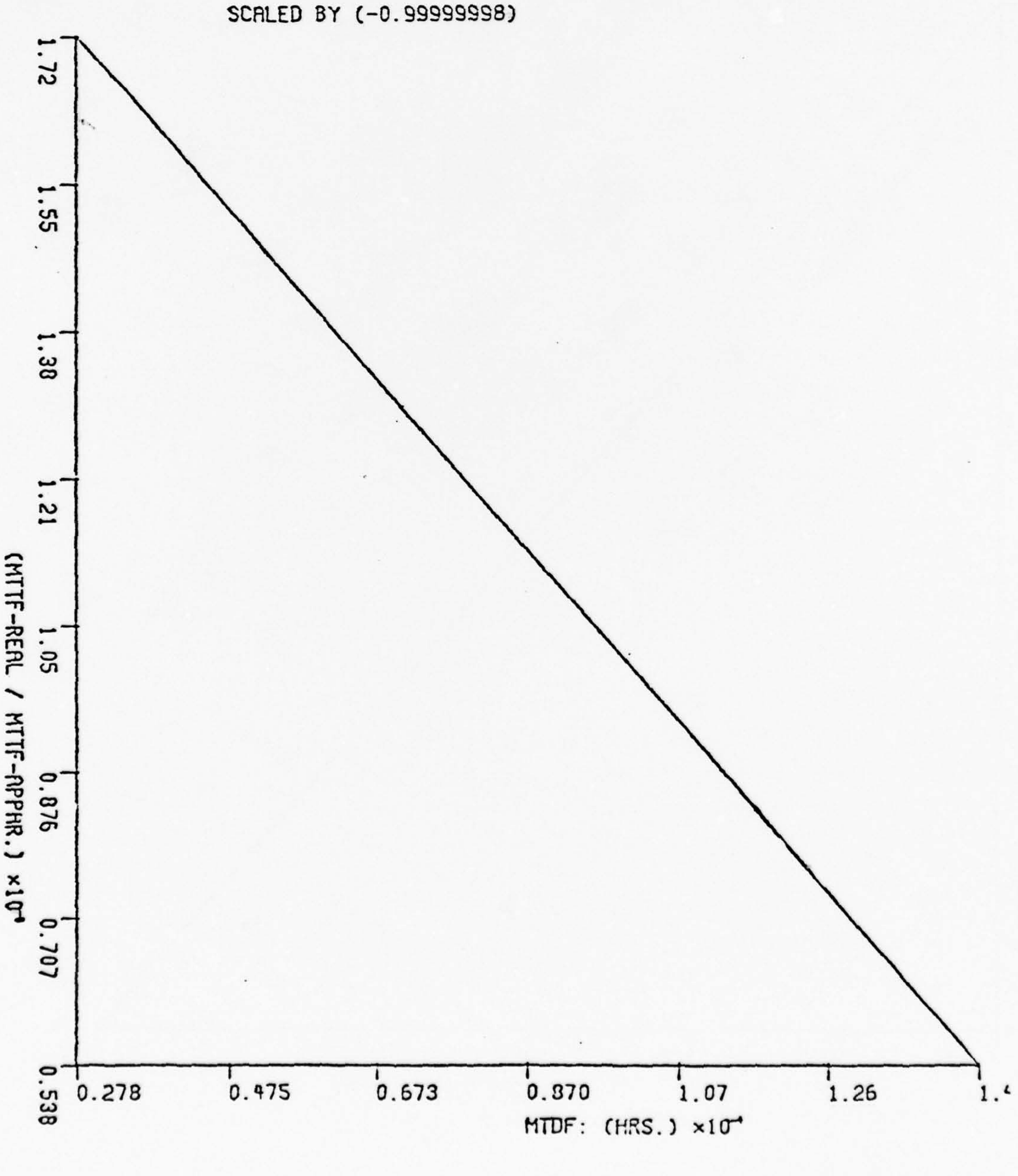


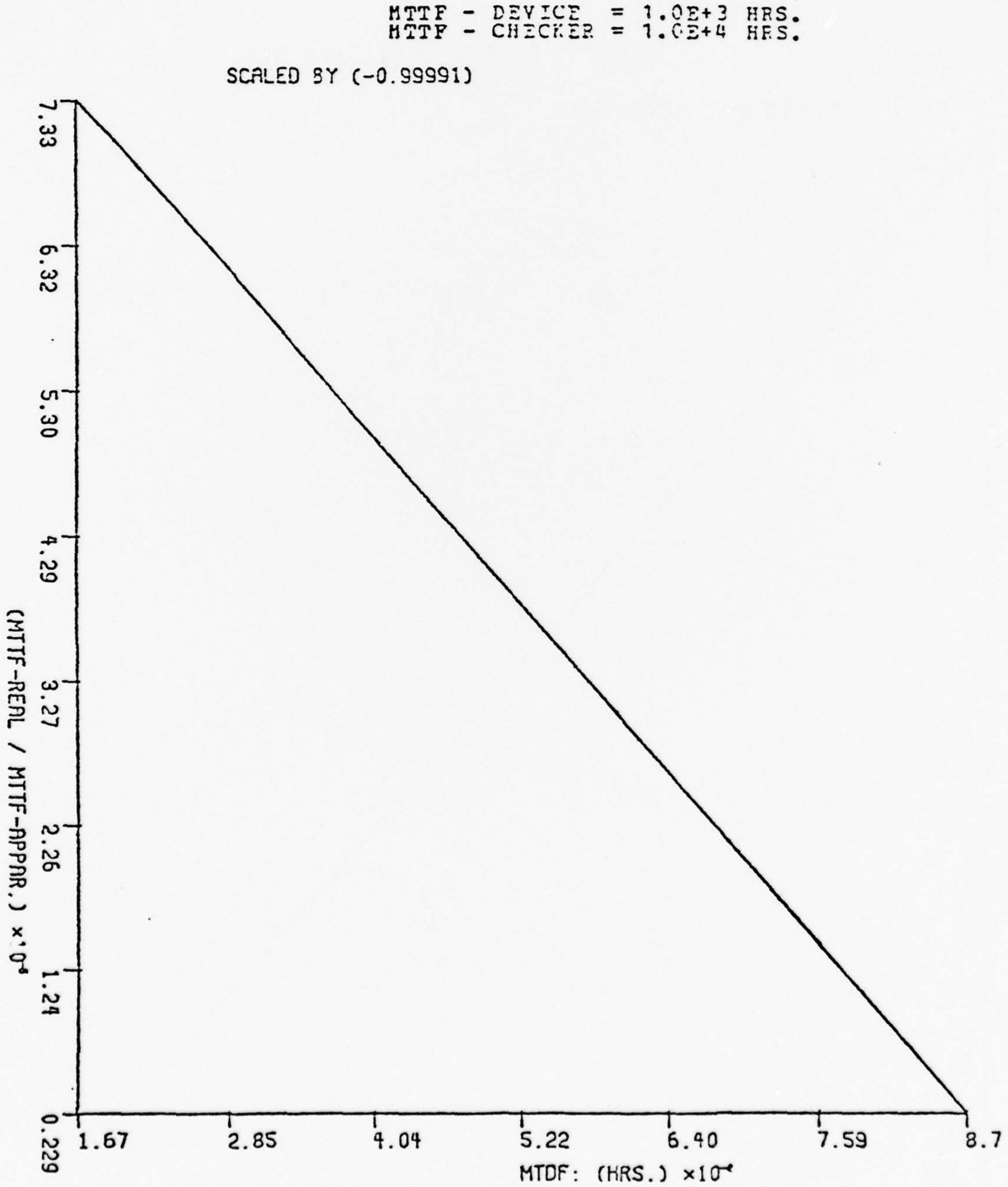


MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A


MTTF - DEVICE = 1.0E+3 HRS.
MTTF - CHECKER = 1.0E+4 HRS.
MTDF 4: (1. -> 10.) MINS.
MTDF 5: (10. -> 20.) MINS.
APPR; MTDF 5
APPR; MTDF 4
REAL; MTDF 4,5
909.36
909.17
909.09
(MTTF)
(HOURS)
1.66
1.91
2.16
2.41
2.66
2.91
3.1
MTDF: (HRS.) ×10⁴

MTTF - DEVICE = 1.0E+4 HRS.
MTTF - CHECKER = 1.0E+5 HRS.




MTTF - DEVICE = 1.0E+3 HRS.
MTTF - CHECKER = 1.0E+4 HRS.
SCALED BY (-0.99991)
7.33
6.32
5.30
4.29
3.27
2.26
1.24
0.229
(MTTF-REAL / MTTF-APPR.) ×10
1.67
2.85
4.04
5.22
6.40
7.59
8.7
MTDF: (HRS.) ×10

```
MTTF - DEVICE  = 1.0E+5 HRS.
MTTF - CHECKER = 1.0E+6 HRS.
MTDF:  0.1 SEC. -> 1.0 SEC.

MTDF                   MTTF REAL               MTTF APPR.
 2.777777777777777E-05  9.090909090909089E+04   9.090909093434340E+04
 3.052503052503052E-05  9.090909090909089E+04   9.090909093684091E+04
 3.387533875338753E-05  9.090909090909089E+04   9.090909093988663E+04
 3.805175038051750E-05  9.090909090909089E+04   9.090909094368337E+04
 4.340277777777777E-05  9.090909090909089E+04   9.090909094854794E+04
 5.050505050505050E-05  9.090909090909089E+04   9.090909095500456E+04
 6.038647342995168E-05  9.090909090909089E+04   9.090909096398766E+04
 7.507507507507507E-05  9.090909090909089E+04   9.090909097734093E+04
 9.920634920634920E-05  9.090909090909089E+04   9.090909099927845E+04
 1.461988304093567E-04  9.090909090909089E+04   9.090909104199889E+04
```

```
MTTF - DEVICE  = 1.0E+5 HRS.
MTTF - CHECKER = 1.0E+6 HRS.
MTDF: 1.0 SEC. -> 10. SEC.

MTDF                       MTTF REAL                  MTTF APPR.
 2.777777777777777E-04      9.090909090909089E+04      9.090909116161611E+04
 3.052503052503052E-04      9.090909090909089E+04      9.090909118659116E+04
 3.387533875338753E-04      9.090909090909089E+04      9.090909121704850E+04
 3.805175038051750E-04      9.090909090909089E+04      9.090909125501588E+04
 4.340277777777777E-04      9.090909090909089E+04      9.090909130366158E+04
 5.050505050505050E-04      9.090909090909089E+04      9.090909136822770E+04
 6.038647342995168E-04      9.090909090909089E+04      9.090909145805881E+04
 7.507507507507507E-04      9.090909090909089E+04      9.090909159159155E+04
 9.920634920634920E-04      9.090909090909089E+04      9.090909181096678E+04
 1.461988304093567E-03      9.090909090909089E+04      9.090909223817115E+04
```

```
MTTF - DEVICE  = 1.0E+5 HRS.
MTTF - CHECKER = 1.0E+6 HRS.
MTDF:  10. MIN. -> 20. MIN.

MTDF                   MTTF REAL              MTTF APPR.
 1.666666666666666E-01  9.090909090909089E+04  9.090924242421714E+04
 1.754385964912280E-01  9.090909090909089E+04  9.090925039869606E+04
 1.851851851851851E-01  9.090909090909089E+04  9.090925925922805E+04
 1.960784313725490E-01  9.090909090909089E+04  9.090926916217536E+04
 2.083333333333333E-01  9.090909090909089E+04  9.090928030299082E+04
 2.222222222222222E-01  9.090909090909089E+04  9.090929292924799E+04
 2.380952380952381E-01  9.090909090909089E+04  9.090930735925579E+04
 2.564102564102564E-01  9.090909090909089E+04  9.090932400926420E+04
 2.777777777777777E-01  9.090909090909089E+04  9.090934343427325E+04
 3.030303030303030E-01  9.090909090909089E+04  9.090936639110106E+04
```

```
MTTF - DEVICE  = 1.0E+5 HRS.
MTTF - CHECKER = 1.0E+6 HRS.
MTDF:  1.0 MIN. -> 10. MIN.

MTDF                     MTTF REAL                MTTF APPR.
 1.666666666666666E-02    9.090909090909089E+04    9.090910606060578E+04
 1.831501831501831E-02    9.090909090909089E+04    9.090910755910722E+04
 2.032520325203252E-02    9.090909090909089E+04    9.090910938654800E+04
 2.283105022831050E-02    9.090909090909089E+04    9.090911166459061E+04
 2.604166666666667E-02    9.090909090909089E+04    9.090911458333268E+04
 3.030303030303031E-02    9.090909090909089E+04    9.090911845729940E+04
 3.623188405797102E-02    9.090909090909089E+04    9.090912384716610E+04
 4.504504504504507E-02    9.090909090909089E+04    9.090913185912999E+04
 5.952380952380958E-02    9.090909090909089E+04    9.090914502164177E+04
 8.771929824561415E-02    9.090909090909089E+04    9.090917065390048E+04
```

```
MTTF - DEVICE  = 1.0E+4 HRS.
MTTF - CHECKER = 1.0E+5 HRS.
MTDF:  0.01 SEC. -> 0.1 SEC

MTDF                     MTTF REAL                MTTF APPR.
 2.777777777777777E-06   9.090909090909090E+03    9.090909093434341E+03
 3.052503052503052E-06   9.090909090909090E+03    9.090909093684092E+03
 3.387533875338753E-06   9.090909090909090E+03    9.090909093988665E+03
 3.805175038051750E-06   9.090909090909090E+03    9.090909094368339E+03
 4.340277777777777E-06   9.090909090909090E+03    9.090909094854797E+03
 5.050505050505049E-06   9.090909090909090E+03    9.090909095500458E+03
 6.038647342995168E-06   9.090909090909090E+03    9.090909096398768E+03
 7.507507507507505E-06   9.090909090909090E+03    9.090909097734095E+03
 9.920634920634920E-06   9.090909090909090E+03    9.090909099927847E+03
 1.461988304093567E-05   9.090909090909090E+03    9.090909104199891E+03
```

```
MTTF - DEVICE  = 1.0E+4 HRS.
MTTF - CHECKER = 1.0E+5 HRS.
MTDF:  0.1 SEC. -> 1.0 SEC.

MTDF                   MTTF REAL               MTTF APPR.
 2.777777777777777E-05  9.090909090909090E+03   9.090909116161614E+03
 3.052503052503052E-05  9.090909090909090E+03   9.090909118659116E+03
 3.387533875338753E-05  9.090909090909090E+03   9.090909121704852E+03
 3.805175038051750E-05  9.090909090909090E+03   9.090909125501589E+03
 4.340277777777777E-05  9.090909090909090E+03   9.090909130366160E+03
 5.050505050505050E-05  9.090909090909090E+03   9.090909136822771E+03
 6.038647342995168E-05  9.090909090909090E+03   9.090909145805882E+03
 7.507507507507507E-05  9.090909090909090E+03   9.090909159159156E+03
 9.920634920634920E-05  9.090909090909090E+03   9.090909181096679E+03
 1.461988304093567E-04  9.090909090909090E+03   9.090909223817116E+03
```

```
MTTF - DEVICE  = 1.0E+4 HRS.
MTTF - CHECKER = 1.0E+5 HRS.
MTDF: 1.0 SEC. -> 10. SEC.

MTDF                           MTTF REAL                     MTTF APPR.
 2.777777777777777E-04           .090909090909090E+03         9.090909343434342E+03
 3.052503052503052E-04          9.090909090909090E+03         9.090909368409365E+03
 3.387533875338753E-04          9.090909090909090E+03         9.090909398866713E+03
 3.805175038051750E-04          9.090909090909090E+03         9.090909436834092E+03
 4.340277777777777E-04          9.090909090909090E+03         9.090909485479793E+03
 5.050505050505050E-04          9.090909090909090E+03         9.090909550045909E+03
 6.038647342995168E-04          9.090909090909090E+03         9.090909639877026E+03
 7.507507507507507E-04          9.090909090909090E+03         9.090909773409766E+03
 9.920634920634920E-04          9.090909090909090E+03         9.090909992784982E+03
 1.461988304093567E-03          9.090909090909090E+03         9.090910419989346E+03
```

```
MTTF - DEVICE  = 1.0E+4 HRS.
MTTF - CHECKER = 1.0E+5 HRS.
MTDF:   1.0 MIN. -> 10. MIN.

MTDF                     MTTF REAL                MTTF APPR.
 1.666666666666666E-02    9.090909090909090E+03    9.090924242421714E+03
 1.831501831501831E-02    9.090909090909090E+03    9.090925740922691E+03
 2.032520325203252E-02    9.090909090909090E+03    9.090927568362835E+03
 2.283105022831050E-02    9.090909090909090E+03    9.090929846404558E+03
 2.604166666666667E-02    9.090909090909090E+03    9.090932765145348E+03
 3.030303030303031E-02    9.090909090909090E+03    9.090936639110107E+03
 3.623188405797102E-02    9.090909090909090E+03    9.090942028973572E+03
 4.504504504504507E-02    9.090909090909090E+03    9.090950040931592E+03
 5.952380952380958E-02    9.090909090909090E+03    9.090963203430992E+03
 8.771929824561415E-02    9.090909090909090E+03    9.090988835655724E+03
```

```
MTTF - DEVICE = 1.0E+4 HRS.
MTTF - CHECKER = 1.0E+5 HRS.
MTDF:   10. MIN. -> 20. MIN.

MTDF                         MTTF REAL                    MTTF APPR.
 1.666666666666666E-01   9.090909090909090E+03   9.091060605808079E+03
 1.754385964912280E-01   9.090909090909090E+03   9.091068580262456E+03
 1.851851851851851E-01   9.090909090909090E+03   9.091077440765680E+03
 1.960784313725490E-01   9.090909090909090E+03   9.091087343679005E+03
 2.083333333333333E-01   9.090909090909090E+03   9.091098484453913E+03
 2.222222222222222E-01   9.090909090909090E+03   9.091111106662176E+03
 2.380952380952381E-01   9.090909090909090E+03   9.091125540610182E+03
 2.564102564102564E-01   9.090909090909090E+03   9.091142190544498E+03
 2.777777777777777E-01   9.090909090909090E+03   9.091161615460157E+03
 3.030303030303030E-01   9.090909090909090E+03   9.091184572167961E+03
```

```
MTTF - DEVICE  = 1.0E+3 HRS.
MTTF - CHECKER = 1.0E+4 HRS.
MTDF:  0.01 SEC. -> 0.1 SEC

MTDF                    MTTF REAL               MTTF APPR.
 2.777777777777777E-06   9.090909090909091E+02   9.090909116161613E+02
 3.052503052503052E-06   9.090909090909091E+02   9.090909118659117E+02
 3.387533875338753E-06   9.090909090909091E+02   9.090909121704852E+02
 3.805175038051750E-06   9.090909090909091E+02   9.090909125501589E+02
 4.340277777777777E-06   9.090909090909091E+02   9.090909130366160E+02
 5.050505050505049E-06   9.090909090909091E+02   9.090909136822771E+02
 6.038647342995168E-06   9.090909090909091E+02   9.090909145805883E+02
 7.507507507507505E-06   9.090909090909091E+02   9.090909159159157E+02
 9.920634920634920E-06   9.090909090909091E+02   9.090909181096679E+02
 1.461988304093567E-05   9.090909090909091E+02   9.090909223817117E+02
```

```
MTTF - DEVICE  = 1.0E+3 HRS.
MTTF - CHECKER = 1.0E+4 HRS.
MTDF:  0.1 SEC. -> 1.0 SEC.

MTDF                      MTTF REAL                 MTTF APPR.
 2.777777777777777E-05     9.090909090909091E+02     9.090909343434342E+02
 3.052503052503052E-05     9.090909090909091E+02     9.090909368409366E+02
 3.387533875338753E-05     9.090909090909091E+02     9.090909398866713E+02
 3.805175038051750E-05     9.090909090909091E+02     9.090909436834092E+02
 4.340277777777777E-05     9.090909090909091E+02     9.090909485479794E+02
 5.050505050505050E-05     9.090909090909091E+02     9.090909550045910E+02
 6.038647342995168E-05     9.090909090909091E+02     9.090909639877026E+02
 7.507507507507507E-05     9.090909090909091E+02     9.090909773409767E+02
 9.920634920634920E-05     9.090909090909091E+02     9.090909992784982E+02
 1.461988304093567E-04     9.090909090909091E+02     9.090910419989346E+02
```

```
MTTF - DEVICE  = 1.0E+3 HRS.
MTTF - CHECKER = 1.0E+4 HRS.
MTDF: 1.0 SEC. -> 10. SEC.

MTDF                    MTTF REAL               MTTF APPR.
 2.777777777777777E-04   9.090909090909091E+02   9.090911616161544E+02
 3.052503052503052E-04   9.090909090909091E+02   9.090911865911781E+02
 3.387533875338753E-04   9.090909090909091E+02   9.090912170485236E+02
 3.805175038051750E-04   9.090909090909091E+02   9.090912550158993E+02
 4.340277777777777E-04   9.090909090909091E+02   9.090913036615989E+02
 5.050505050505050E-04   9.090909090909091E+02   9.090913682277085E+02
 6.038647342995168E-04   9.090909090909091E+02   9.090914580588161E+02
 7.507507507507507E-04   9.090909090909091E+02   9.090915915915402E+02
 9.920634920634920E-04   9.090909090909091E+02   9.090918109667212E+02
 1.461988304093567E-03   9.090909090909091E+02   9.090922381709910E+02
```

```
MTTF - DEVICE  = 1.0E+3 HRS.
MTTF - CHECKER = 1.0E+4 HRS.
MTDF:  1.0 MIN. -> 10. MIN.

MTDF                         MTTF REAL                  MTTF APPR.
 1.666666666666666E-02        9.090909090909091E+02      9.091060605808079E+02
 1.831501831501831E-02        9.090909090909091E+02      9.091075590770645E+02
 2.032520325203252E-02        9.090909090909091E+02      9.091093865108551E+02
 2.283105022831050E-02        9.090909090909091E+02      9.091116645437297E+02
 2.604166666666667E-02        9.090909090909091E+02      9.091145832716817E+02
 3.030303030303031E-02        9.090909090909091E+02      9.091184572167962E+02
 3.623188405797102E-02        9.090909090909091E+02      9.091238470479848E+02
 4.504504504504507E-02        9.090909090909091E+02      9.091318589474002E+02
 5.952380952380958E-02        9.090909090909091E+02      9.091450213229249E+02
 8.771929824561415E-02        9.090909090909091E+02      9.091706532079860E+02
```

```
MTTF - DEVICE  = 1.0E+3 HRS.
MTTF - CHECKER = 1.0E+4 HRS.
MTDF:   10. MIN. -> 20. MIN.

MTDF                    MTTF REAL               MTTF APPR.
 1.666666666666666E-01   9.090909090909091E+02   9.092424217172136E+02
 1.754385964912280E-01   9.090909090909091E+02   9.092503959260681E+02
 1.851851851851851E-01   9.090909090909091E+02   9.092592561417211E+02
 1.960784313725490E-01   9.090909090909091E+02   9.092691587152478E+02
 2.083333333333333E-01   9.090909090909091E+02   9.092802990846780E+02
 2.222222222222222E-01   9.090909090909091E+02   9.092929248035911E+02
 2.380952380952381E-01   9.090909090909091E+02   9.093073541539053E+02
 2.564102564102564E-01   9.090909090909091E+02   9.093240033472334E+02
 2.777777777777777E-01   9.090909090909091E+02   9.093434273290386E+02
 3.030303030303030E-01   9.090909090909091E+02   9.093663828368835E+02
```

```
MTTF - DEVICE  = 1.0E+5 HRS.
MTTF - CHECKER = 1.0E+6 HRS.
MTDF:  0.01 SEC. -> 0.1 SEC


MTDF                       RATIO: (REAL / APPAR.)
 2.777777777777777E-06      9.999999999722224E-01
 3.052503052503052E-06      9.999999999694750E-01
 3.387533875338753E-06      9.999999999661248E-01
 3.805175038051750E-06      9.999999999619484E-01
 4.340277777777777E-06      9.999999999565974E-01
 5.050505050505049E-06      9.999999999494950E-01
 6.038647342995168E-06      9.999999999396136E-01
 7.507507507507505E-06      9.999999999249249E-01
 9.920634920634920E-06      9.999999999007937E-01
 1.461988304093567E-05      9.999999998538014E-01
```

```
MTTF - DEVICE  = 1.0E+4 HRS.
MTTF - CHECKER = 1.0E+5 HRS.
MTDF: 1.0 SEC. -> 10. SEC.

MTDF                          RATIO: (REAL / APPAR.)
 2.777777777777777E-04         9.999999722222230E-01
 3.052503052503052E-04         9.999999694749705E-01
 3.387533875338753E-04         9.999999661246625E-01
 3.805175038051750E-04         9.999999619482512E-01
 4.340277777777777E-04         9.999999565972244E-01
 5.050505050505050E-04         9.999999494949523E-01
 6.038647342995168E-04         9.999999396135307E-01
 7.507507507507507E-04         9.999999249249312E-01
 9.920634920634920E-04         9.999999007936617E-01
 1.461988304093567E-03         9.999998538011931E-01
```

```
MTTF - DEVICE  = 1.0E+3 HRS.
MTTF - CHECKER = 1.0E+4 HRS.
MTDF:  10. MIN. -> 20. MIN.

MTDF                          RATIO: (REAL / APPAR.)
 1.666666666666666E-01         9.998333638832881E-01
 1.754385964912280E-01         9.998245952535476E-01
 1.851851851851851E-01         9.998148525300401E-01
 1.960784313725490E-01         9.998039638509342E-01
 2.083333333333333E-01         9.997917143987836E-01
 2.222222222222222E-01         9.997778320854902E-01
 2.380952380952381E-01         9.997619671038537E-01
 2.564102564102564E-01         9.997436620440388E-01
 2.777777777777777E-01         9.997223070728390E-01
```

APPENDIX III

THE SIMULATION

```
FINAL STATISTICS

SYSTEM PARAMETERS

   LAMBDA                          9.999999999999999E-05
   DELTA                           3.600000000000000E+03
   MU                              1.000000000000000E+00
   ALPHA                           9.999999999999998E-06
   BETA                            2.000000000000000E+00

MTTF - DEVICE   = 1.0E+4 HRS.

MTTF - CHECKER = 1.0E+5 HRS.

MTTR - DEVICE   = 1.0 HRS.

MTTR - CHECKER = 0.5  HRS.

MTDF = 1.0 SECS.

MODEL REAL AVAILABILITY                                      9.998949932518990E-01
SIMULATION REAL AVAILABILITY                                 9.999144755230651E-01
   DIFFERENCE BETWEEN MODEL & SIMULATE                       1.949227116700147E-05
   VARIANCE OF SIMULATION                                    1.245908003226940E-03

MODEL APPR AVAILABILITY                                      9.998950110267587E-01
SIMULATION APPR AVAILABILITY                                 9.999144965202763E-01
   DIFFERENCE BETWEEN MODEL & SIMULATE                       1.948549351758754E-05
   VARIANCE OF SIMULATION                                    1.245678326293514E-03

ALL AVERAGES ARE FOR AN AVERAGE CYCLE TIME

AVG CYCLE TIME                                               9.817177712253395E+03

PROB OF BEING IN SYS WORKING STATE                           9.999144755230651E-01
   AVG TIME IN SYS WORKING STATE                             9.816338103264570E+03
   VARIANCE OF SYS WORKING STATE                             1.245908003226940E-03

PROB OF BEING IN SYS FAILURE STATE                           2.099721124442769E-08
   AVG TIME IN SYS FAILURE STATE                             2.061333542482717E-04
   VARIANCE OF SYS FAILURE STATE                             4.358023880517725E-08

PROB OF BEING IN SYS DETECT STATE                            8.440311127473104E-05
   AVG TIME IN SYS DETECT STATE                              8.286003428511320E-01
   VARIANCE OF SYS DETECT STATE                              7.041775184353533E-01

PROB OF BEING IN CHK FAILURE STATE                           1.100368449318776E-06
   AVG TIME IN CHK FAILURE STATE                             1.080251261591911E-02
   VARIANCE OF CHK FAILURE STATE                             1.196961745851276E-04
```

```
FINAL STATISTICS

SYSTEM PARAMETERS

  LAMBDA                     9.999999999999999E-04
  DELTA                      6.000000000000000E+00
  MU                         5.000000000000000E-01
  ALPHA                      9.999999999999999E-05
  BETA                       1.000000000000000E+00

MTTF - DEVICE  = 1.0E+3 HRS.

MTTF - CHECKER = 1.0E+4 HRS.

MTTR - DEVICE  = 2.0 HRS.

MTTR - CHECKER = 1.0  HRS.

MTDF = 10.0 MINS.

MODEL REAL AVAILABILITY                                 9.977384594918187E-01
SIMULATION REAL AVAILABILITY                            9.981671555842278E-01
  DIFFERENCE BETWEEN MODEL & SIMULATE                   4.286960924091975E-04
  VARIANCE OF SIMULATION                                2.075378777995467E-02

MODEL APPR AVAILABILITY                                 9.979947492350673E-01
SIMULATION APPR AVAILABILITY                            9.992229136997908E-01
  DIFFERENCE BETWEEN MODEL & SIMULATE                   2.881694664721531E-04
  VARIANCE OF SIMULATION                                1.973639430673568E-02

ALL AVERAGES ARE FOR AN AVERAGE CYCLE TIME

AVG CYCLE TIME                                          9.834362960499396E+02

PROB OF BEING IN SYS WORKING STATE                      9.981671555842279E-01
  AVG TIME IN SYS WORKING STATE                         9.816338103264570E+02
  VARIANCE OF SYS WORKING STATE                         2.075378777995467E-02

PROB OF BEING IN SYS FAILURE STATE                      1.257631155629856E-04
  AVG TIME IN SYS FAILURE STATE                         1.236800125489631E-01
  VARIANCE OF SYS FAILURE STATE                         1.568321253119840E-02

PROB OF BEING IN SYS DETECT STATE                       1.685112393426743E-03
  AVG TIME IN SYS DETECT STATE                          1.657200685702264E+00
  VARIANCE OF SYS DETECT STATE                          2.815175829968516E+00

PROB OF BEING IN CHK FAILURE STATE                      2.196891178271205E-05
  AVG TIME IN CHK FAILURE STATE                         2.160502523183822E-02
  VARIANCE OF CHK FAILURE STATE                         4.786400737387820E-04
```

```
FINAL STATISTICS

SYSTEM PARAMETERS

  LAMBDA                     9.9999999999999999E-05
  DELTA                      3.6000000000000000E+03
  ALPHA                      9.9999999999999998E-06

MTTF - DEVICE  = 1.0E+4 HRS.

MTTF - CHECKER = 1.0E+5 HRS.

MTDF = 1.0 SEC.


MODEL REAL MTTF              9.0909090909090909E+03
SIMULATION REAL MTTF         8.4654877251014111E+03
  DIFFERENCE BETWEEN MODEL & SIMULATE                   6.2542135580767965E+02
  VARIANCE OF SIMULATION                                7.1681391307970031E+07


MODEL APPR MTTF              9.090909343434342E+03
SIMULATION APPR MTTF         8.465487973613243E+03
  DIFFERENCE BETWEEN MODEL & SIMULATE                   6.2542137982109955E+02
  VARIANCE OF SIMULATION                                7.168139173997958E+07


PROB OF BEING IN SYS WORKING STATE                      9.9999997182538945E-01

PROB OF BEING IN SYS FAILURE STATE                      2.817461128228426E-08
```

```
FINAL STATISTICS.

SYSTEM PARAMETERS

  LAMBDA                        9.999999999999999E-04
  DELTA                         6.000000000000000E+00
  ALPHA                         9.999999999999999E-05

MTTF - DEVICE  = 1.0E+3 HRS.

MTTF - CHECKER = 1.0E+4 HRS.

MTDF = 10.0 MINS.


MODEL REAL MTTF                 9.090909090909091E+02
SIMULATION REAL MTTF            8.465487725101411E+02
  DIFFERENCE BETWEEN MODEL & SIMULATE                        6.254213558076798E+01
  VARIANCE OF SIMULATION                                     7.168139130797040E+05


MODEL APPR MTTF                 9.092424217172138E+02
SIMULATION APPR MTTF            8.466918795097548E+02
  DIFFERENCE BETWEEN MODEL & SIMULATE                        6.255054210745896E+01
  VARIANCE OF SIMULATION                                     7.168398579549755E+05


PROB OF BEING IN SYS WORKING STATE                           9.998309803998294E-01

PROB OF BEING IN SYS FAILURE STATE                           1.690191001710734E-04
```

```
TRIAL: PROCEDURE OPTIONS(MAIN) ;
/*                                                                    */
/*  SIMULATION OF THE MAINTAINED SYSTEM                               */
/*                                                                    */
/*  INPUT ON A CARD :  LAMBDA, DELTA, MU, ALPHA, BETA                 */
/*                                                                    */
 DECLARE (MODEL_REAL_AVAIL, MODEL_APPR_AVAIL,
          SYS_FAILED_TIME, CHK_FAILED_TIME,
          SYS_REPAIR_TIME, CHK_REPAIR_TIME,
          SYS_DETECT_TIME,
          SIMUL_REAL_VAR, SIMUL_APPR_VAR,
          SYS_WORKING_STATE, SYS_FAILURE_STATE,
          SYS_DETECT_STATE,  CHK_FAILURE_STATE,
          SYS_WORKING_VAR, SYS_FAILURE_VAR,
          SYS_DETECT_VAR, CHK_FAILURE_VAR,
          REAL_AVAIL_TIME, PREV_REAL_AVAIL,
          APPR_AVAIL_TIME, PREV_APPR_AVAIL,
          CHANGE_SIMUL_REAL, REAL_MOD_SIM_DIFF,
          CHANGE_SIMUL_APPR, APPR_MOD_SIM_DIFF,
          DIFF_REAL_AVAIL, DIFF_APPR_AVAIL,
          PREV_SIMUL_APPR_AVAIL, NOW_SIMUL_APPR_AVAIL,
          PREV_SIMUL_REAL_AVAIL, NOW_SIMUL_REAL_AVAIL,
          FINAL_REAL_DIFF, VAR_SIMUL_REAL,
          FINAL_APPR_DIFF, VAR_SIMUL_APPR,
          AVG_SYS_WORKING, VAR_MODEL_REAL,
          AVG_SYS_FAILURE, VAR_MODEL_APPR,
          AVG_SYS_DETECT, AVG_CHK_FAILURE,
          AVG_APPR_AVAIL, AVG_CYCLE_TIME,
          PROB_SYS_WORKING, PROB_SYS_FAILURE,
          PROB_SYS_DETECT,  PROB_CHK_FAILURE,
          VAR_SYS_WORKING, VAR_SYS_DETECT,
          VAR_SYS_FAILURE, VAR_CHK_FAILURE,
          RUN_WORKING_AVG, RUN_FAILED_AVG,
          RUN_DETECTED_AVG, RUN_CHECKER_AVG,
          AVG_CYCLE_REAL, AVG_CYCLE_APPR,
          COUNTER, CLOCK, CYCLE_TIME,
          LAMBDA, DELTA, MU, ALPHA, BETA)
          DECIMAL FLOAT(16) INITIAL(0.0) ;
/*                                                                    */
/*  DECLARE THE CONTROL VARIABLES                                     */
/*                                                                    */
 DECLARE (ON,
          SELECT,
          RESET,
          EXEC,
          STAT,
          DUMP,
          LOOKING,
          REATTAIN,
          SIMULATE) FIXED INITIAL (0) ;
```

```
/*                                                                    */
/*                                                                    */
/*                                                                    */
/*                                                                    */
/*  VARGEN: TAUSWORTHE -.TYPE PSEUDO RANDOM NUMBER.GENERATOR          */
/*          DOCUMENT NC. LS-110-1                                     */
/*                                                                    */
 DECLARE DIST FIXED BINARY(15,0),
         SEED FIXED BINARY(31,0),
         FLAG FIXED BINARY(15,0),
         (LR, DR, MR, AR, BR, A, B, DUM) FLOAT(6) ;
/*                                                                    */
 DECLARE
         VARGEN ENTRY(FIXED BINARY(15,0),
                      FIXED BINARY(31,0),
                      FIXED BINARY(15,0),
                      FLOAT(6), FLOAT(6),
                      FLOAT(6), FLOAT(6) )
                RETURNS(FLOAT(6)) ;
/*                                                                    */
/*  VARGEN PARAMETERS:                                                */
/*                                                                    */
    DIST = 2 ;
    SEED = 1010101010101010101010101010101B ;
    A = 0.0 ;
    B = 0.999 ;
/*                                                                    */
/*                                                                    */
/*                                                                    */
/*  GET SYSTEM PARAMETERS: RATES IN (1/HRS.)                          */
/*                                                                    */
    GET LIST(LAMBDA, DELTA, MU, ALPHA, BETA) ;
/*                                                                    */
/*                                                                    */
/*  EQUATIONS OF THE MODEL FOR                                        */
/*     REAL AND APPARANT AVAILABILITY                                 */
/*                                                                    */
  MODEL_REAL_AVAIL = (1.0) /
       (1.0 + (LAMBDA/MU) + (LAMBDA/DELTA) + (ALPHA/BETA)) ;
  MODEL_APPR_AVAIL = MODEL_REAL_AVAIL +
                      (LAMBDA/DELTA)*(MODEL_REAL_AVAIL) ;
/*                                                                    */
    ON = 1 ;
    SELECT = 1 ;
/*                                                                    */
/*                                                                    */
```

```
L0: CO WHILE (CN = 1) ;
/*                                                                       */
/*  MASTER PROGRAM CCNTROL                                               */
/*                                                                       */
/*                                                                       */
/*                                                                       */
L1:  DO WHILE (SELECT = 1) ;
 /*                                                                      */
/*  THE 'SELECT' MCDULE RCUTINE                                          */
/*     TEST CCNDITICNS, AND SET APPROPRIATE CCNTROLS                     */
 /*                                                                      */
/*                                                                       */
/*                                                                       */
/*                                                                       */
/*  IF LCCKING FOR THE STEADY STATE,                                     */
/*     THEN INCREMENT 'LOOKING', AND 'EXEC' THE CYCLE RCUTINE            */
/*                                                                       */
        IF (LCCKING >= 0) &
           (LCOKING <= 10)
         THEN
            DO;
               LOOKING = LOOKING + 1 ;
               REATTAIN = -1 ;
               SIMULATE = -1 ;
               ON = 1 ;
               SELECT = 0 ;
               RESET = 0 ;
               EXEC = 1 ;
               STAT = 0 ;
               DUMP = 0 ;
               PUT PAGE;
               PUT SKIP LIST('LOOKING FOR THE STEADY STATE') ;
            END;
/*                                                                       */
/*                                                                       */
/*                                                                       */
/* IF THE STEADY STATE HAS BEEN FOUND,                                   */
/*    RESET ALL VARIABLES TO 0, AND RE-ATTAIN THE STEADY STATE           */
/*                                                                       */
        IF (LOOKING = 11) &
           (DIFF_REAL_AVAIL < 0.01) &
           (DIFF_APPR_AVAIL < 0.01)
         THEN
            DO;
              LOOKING = -1 ;
              REATTAIN = -1 ;
              SIMULATE = -1 ;
              ON = 1 ;
              SELECT = 0 ;
              RESET = 1 ;

              EXEC = 0 ;
              STAT = 0 ;
              DUMP = 0 ;
              PUT SKIP(2) LIST('THE STEADY STATE HAS BEEN REACHED') ;
              PUT SKIP(2) LIST('THE REAL AVAIL FOR THE FIRST 05 RUNS',
                               PREV_SIMUL_REAL_AVAIL) ;
              PUT SKIP(2) LIST('THE REAL AVAIL FOR THE FIRST 10 RUNS',
                               NOW_SIMUL_REAL_AVAIL) ;
              PUT SKIP(2) LIST('THE DIFFERENCE =', DIFF_REAL_AVAIL) ;
              PUT SKIP(4) LIST('THE APPR AVAIL FOR THE FIRST 05 RUNS',
                               PREV_SIMUL_APPR_AVAIL) ;
              PUT SKIP(2) LIST('THE APPR AVAIL FOR THE FIRST 10 RUNS',
                               NOW_SIMUL_APPR_AVAIL) ;
              PUT SKIP(2) LIST('THE DIFFERENCE =', DIFF_APPR_AVAIL) ;
            END;
/*                                                                       */
/*                                                                    */
/*                                                                    */
/*  IF THE STEADY STATE WAS NOT REACHED,                              */
/*    PRINT STATISTICS, AND END                                       */
/*                                                                    */
       IF (LOOKING = 11) &
          ( (DIFF_REAL_AVAIL >= 0.01) | (DIFF_APPR_AVAIL >= 0.01) )
        THEN
           DO ;
             LOOKING = -1 ;
             REATTAIN = -1 ;
             SIMULATE = -1 ;
             CN = 0 ;
             SELECT = 0 ;
             RESET = 0 ;
             EXEC = 0 ;
             STAT = 1 ;
             DUMP = 1 ;
             PUT PAGE ;
             PUT SKIP(2) LIST('FAILED TO REACH STEADY STATE') ;
             PUT SKIP(2) LIST('DIFF_REAL_AVAIL', DIFF_REAL_AVAIL) ;
             PUT SKIP(2) LIST('DIFF_APPR_AVAIL', DIFF_APPR_AVAIL) ;
           END;
/*                                                                    */
/*                                                                    */
/*                                                                    */
/*                                                                    */
/*                                                                    */
/*                                                                    */
/*                                                                    */
/*                                                                    */
/*                                                                    */
/*                                                                    */
/*                                                                    */
/*  IF RE-ATTAINING THE STEADY STATE,                                          */
/*    THEN INCREMENT 'REATTAIN', AND 'EXEC' THE CYCLE ROUTINE                  */
/*                                                                              */
        IF (REATTAIN >= 0) &
           (REATTAIN <= 10)
         THEN
            DO;
               LOOKING = -1 ;
               REATTAIN = REATTAIN + 1 ;
               SIMULATE = -1 ;
               SELECT = 0 ;
               RESET = 0 ;
               EXEC = 1 ;
               STAT = 0 ;
               DUMP = 0 ;
               PUT PAGE ;
               PUT SKIP LIST('RE-ATTAINING THE STEADY STATE') ;
            END;
/*                                                                              */
/*                                                                              */
/*                                                                              */
/*  IF THE STEADY STATE HAS BEEN RE-ATTAINED,                                  */
/*    THEN BEGIN THE SIMULATION                                                */
/*                                                                              */
        IF (REATTAIN = 11)
         THEN
            DO;
               LOOKING = -1 ;
               REATTAIN = -1 ;
               SIMULATE = 0 ;
               ON = 1 ;
               SELECT = 0 ;
               RESET = 0 ;
               EXEC = 1 ;
               STAT = 0 ;
               DUMP = 0 ;
               PUT SKIP(2) LIST('THE SIMULATION NOW BEGINS') ;
            END;
/*                                                                              */
/*                                                                              */
/*                                                                              */
/*                                                                              */
/* IF IN THE SIMULATICN,                                                      */
/*     THEN INCREMENT 'SIMULATE', AND 'EXEC'THE CYCLE ROUTINE                  */
/*                                                                            */
        IF (SIMULATE >= 0) &
           (SIMULATE <= 30)
         THEN
            DO;
              LOOKING = -1 ;
              REATTAIN = -1 ;
              SIMULATE = SIMULATE + 1 ;
              CN = 1 ;
              SELFCT = 0 ;
              RESET = 0 ;
              EXEC = 1 ;
              STAT = C ;
              DUMP = C ;
              PUT PAGE ;
              PUT SKIP(2) LIST('SIMULATICN') ;
            END;
/*                                                                            */
/*                                                                            */
/*                                                                            */
/*                                                                            */
/*                                                                            */
/*                                                                            */
/*                                                                            */
/*                                                                            */
/*                                                                            */
/*   IF SIMULATION IS CCMPLETED,                                              */
/*     THEN PRINT STATISTICS AND END                                          */
 /*                                                                           */
        IF (SIMULATE = 31)
         THEN
            DO;
              LCOKING = -1 ;
              REATTAIN = -1 ;
              SIMULATE = -1 ;
              CN = 1 ;
              SELECT = 0 ;
              RESET = 0 ;
              EXEC = 0 ;
              STAT = 1 ;
              DUMP = 0 ;
              PUT PAGE ;
              PUT SKIP(2) LIST('FINAL STATISTICS') ;
            END;
 /*                                                                           */
 /*                                                                           */
 /*                                                                           */
END L1,
/*                                                                         */
/*                                                                         */
/*                                                                         */
L2: DO WHILE (RESET = 1);
/*                                                                         */
/*   RESET ACCUMULATING VARIABLES BACK TO 0                                */
/*                                                                         */
        CLOCK = C.C ;
        COUNTER = 0.0 ;
        SYS_WORKING_STATE = 0.0 ;
        SYS_FAILURE_STATE = 0.0 ;
        SYS_DETECT_STATE = 0.0 ;
        CHK_FAILURE_STATE = 0.0 ;
        SYS_WORKING_VAR = 0.0 ;
        SYS_FAILURE_VAR = 0.0 ;
        SYS_DETECT_VAR = 0.0 ;
        CHK_FAILURE_VAR = 0.0 ;
        SIMUL_REAL_VAR = 0.0 ;
        SIMUL_APPR_VAR = 0.0 ;
        RUN_WORKING_AVG = C.0 ;
        RUN_FAILED_AVG = 0.0 ;
        RUN_DETECTED_AVG = 0.0 ;
        RUN_CHECKER_AVG = 0.0 ;
        RESET = 0 ;
        SELECT = 1 ;
        REATTAIN = 0 ;
/*                                                                         */
/*                                                                         */
     END L2;
/*                                                                         */
/*                                                                         */
/*                                                                         */
/*                                                                         */
L3: DO WHILE (EXEC = 1) ;
/*                                                                         */
/*   THE 'EXEC' CYCLE ROUTINE                                              */
/*                                                                         */
/*                                                                         */
/*   DETERMINE: TIME TO SYSTEM FAILURE  (IE._UNIT) AND                     */
/*                TIME TO CHECKER FAILURE                                  */
/*                                                                         */
 CCUNTER = CCUNTER + 1.0 ;
     PUT SKIP LIST('CCUNTER', COUNTER) ;
/*                                                                         */
 CALL VARGEN(DIST, SEED, FLAG, A, B, LR, DUM) ;
    SYS_FAILED_TIME = -(1.0/LAMBDA)*(LOG(LR)) ;
 CALL VARGEN(DIST, SEED, FLAG, A, B, AR, DUM) ;
    CHK_FAILED_TIME = -(1.0/ALPHA)*(LOG(AR)) ;
/*                                                                         */
/*                                                                          */
 IF (CHK_FAILED_TIME > SYS_FAILED_TIME)
   THEN DO ;
/*                                                                          */
/*                                                                          */
/*   IF THE SYSTEM FAILS FIRST ...                                          */
/*                                                                          */
/*                                                                          */
/*     IN 'W' : THE WCRKING STATE                                           */
/*                                                                          */
/*                                                                          */
       CLCCK = CLCCK + SYS_FAILEC_TIME;
       CYCLE_TIME = SYS_FATLED_TIME ;
       APPR_AVAIL_TIME = SYS_FAILEC_TIME ;
       REAL_AVAIL_TIME = SYS_FAILEC_TIME ;
       SYS_WORKING_STATE = SYS_WORKING_STATE + SYS_FAILED_TIME ;
/*                                                                          */
         PUT SKIP(2) LIST('THE SYSTEM (UNIT) FAILS FIRST') ,
         PUT SKIP LIST('*******************************') ;
         PUT SKIP(2) LIST('IN THE W - STATE  (WORKING)') ;
         PUT SKIP(2) LIST('CLCCK', CLCCK) ;
         PUT SKIP LIST('CYCLE_TIME', CYCLE_TIME) ;
         PUT SKIP LIST('SYS_FAILED_TIME', SYS_FAILED_TIME) ;
         PUT SKIP LIST('    APPR_AVAIL_TIME', APPR_AVAIL_TIME) ;
         PUT SKIP LIST('    REAL_AVAIL_TIME', REAL_AVAIL_TIME) ;
         PUT SKIP LIST('    SYS_WORKING_STATE', SYS_WORKING_STATE) ;
         PUT SKIP LIST('    SYS_WORKING_VAR', SIMUL_REAL_VAR) ;
/*                                                                          */
/*                                                                          */
/*     DETERMINE THE TIME TO DETECTION                                      */
/*                                                                          */
     CALL VARGEN(DIST, SEED, FLAG, A, B, DR, CUM) ;
     SYS_DETECT_TIME = -(1.0/DELTA)*(LOG(DR)) ;
/*                                                                          */
/*                                                                          */
/*     IN 'F' : THE FAILURE STATE (UNDETECTED)                              */
/*                                                                          */
/*                                                                          */
       CLCCK = CLCCK + SYS_DETECT_TIME ;
       CYCLE_TIME = CYCLE_TIME + SYS_DETECT_TIME ;
       APPR_AVAIL_TIME = APPR_AVAIL_TIME + SYS_DETECT_TIME ;
       SYS_FAILURE_STATE = SYS_FAILURE_STATE + SYS_DETECT_TIME ;
/*                                                                          */
         PUT SKIP(2) LIST('IN THE F- STATE (FAILED)') ;
         PUT SKIP(2) LIST('CLOCK', CLOCK) ;
         PUT SKIP LIST('CYCLE_TIME', CYCLE_TIME) ;
         PUT SKIP LIST('SYS_DETECT_TIME', SYS_DETECT_TIME) ;
         PUT SKIP LIST('    APPR_AVAIL_TIME', APPR_AVAIL_TIME) ;
         PUT SKIP LIST('    SYS_FAILURE_STATE', SYS_FAILURE_STATE) ;
         PUT SKIP  LIST('    SYS_FAILURE_VAR', SYS_FAILURE_VAR) ;
```

```
/*                                                                                      */
/*                                                                                      */
/*     DETERMINE THE TIME TO SYSTEM REPAIR                                              */
/*                                                                                      */
    CALL VARGEN(DIST, SEED, FLAG, A, B, MR, DUM) ;
      SYS_REPAIR_TIME = -(1.0/MU)*(LOG(MR)) ;
/*                                                                                      */
/*                                                                                      */
/*                                                                                      */
/*     IN 'D' : THE DETECTED FAILURE STATE                                              */
/*                                                                                      */
/*                                                                                      */
       CLOCK = CLOCK + SYS_REPAIR_TIME ;
       CYCLE_TIME = CYCLE_TIME + SYS_REPAIR_TIME ;
       SYS_DETECT_STATE = SYS_DETECT_STATE + SYS_REPAIR_TIME ;
/*                                                                                      */
         PUT SKIP(2) LIST('IN THE D - STATE (CETECTED)') ;
         PUT SKIP(2) LIST('CLOCK', CLOCK) ;
         PUT SKIP LIST('CYCLE_TIME', CYCLE_TIME) ;
         PUT SKIP LIST('SYS_REPAIR_TIME', SYS_REPAIR_TIME) ;
         PUT SKIP LIST('    SYS_DETECT_STATE', SYS_DETECT_STATE) ;
         PUT SKIP LIST('    SYS_DETECT_VAR', SYS_DETECT_VAR) ;
/*                                                                                      */
   END;
/*                                                                                      */
/*                                                                                      */
/*                                                                                      */
/*                                                                                      */
/*                                                                                      */
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
/* IF THE CHECKER FAILS FIRST....                                                   */
/*                                                                                  */
/*                                                                                  */
   ELSE DO ;
/*                                                                                  */
/*                                                                                  */
/*     IN 'W' : THE WORKING STATE                                                   */
/*                                                                                  */
       CLCCK = CLCCK + CHK_FAILED_TIME :
       CYCLE_TIME = CHK_FATLED_TIMF ;
       APPR_AVAIL_TIME = CHK_FAILED_TIME ;
       REAL_AVAIL_TIME = CHK_FAILED_TIME ;
       SYS_WORKING_STATE = SYS_WORKING_STATE + CHK_FAILED_TIME ;
/*                                                                                  */
         PUT SKIP(2) LIST('THE CHECKER FAILS FIRST') ;
         PUT SKIP LIST('*****************************') ;
         PUT SKIP(2) LIST('IN THE W - STATE  (WORKING)') ;
         PUT SKIP(2) LIST('CLOCK', CLOCK) ;
         PUT SKIP LIST('CYCLE_TIME', CYCLE_TIME) ;
         PUT SKIP LIST('CHK_FAILED_TIME', CHK_FAILED_TIME) ,
         PUT SKIP LIST('    APPR_AVAIL_TIME', APPR_AVAIL_TIME) ;
         PUT SKIP LIST('    REAL_AVAIL_TIME', REAL_AVAIL_TIME) ;
         PUT SKIP LIST('    SYS_WORKING_STATE', SYS_WORKING_STATE) ;
         PUT SKIP LIST('    SYS_WORKING_VAR', SIMUL_REAL_VAR) ;
/*                                                                                  */
/*                                                                                  */
/*     DETERMINE THE TIME TC CHECKER REPAIR                                         */
/*                                                                                  */
     CALL VARGEN(DIST, SEED, FLAG, A, B, BR, CUM) ;
       CHK_REPAIR_TIME = -(1.0/BETA)*(LOG(BR)) ;
/*                                                                                  */
/*                                                                                  */
/*                                                                                  */
/*     IN 'C' : THE CHECKER REPAIR STATE                                            */
/*                                                                                  */
/*                                                                                  */
       CLCCK = CLCCK + CHK_REPAIR_TIME :
       CYCLE_TIME = CYCLF_TIME + CHK_REPAIR_TIME ;
       CHK_FAILURE_STATE = CHK_FAILURE_STATE + CHK_REPAIR_TIME ;
/*                                                                                  */
         PUT SKIP(2) LIST('IN THE C - STATE (CHECKER REPAIR)') ;
         PUT SKIP(2) LIST('CLOCK', CLOCK) ;
         PUT SKIP LIST('CYCLE_TIME', CYCLE_TIME) ;
         PUT SKIP LIST('CHK_REPAIR_TIME', CHK_REPAIR_TIME) ;
         PUT SKIP LIST('    CHK_FAILURE_STATE', CHK_FAILURE_STATE) ;
         PUT SKIP LIST('    CHK_FAILURE_VAR', CHK_FAILURE_VAR) ;
/*                                                                                  */
/*                                                                                  */
   END;
/*                                                                                  */
/*  CYCLE_STATISTICS                                                  */
/*                                                                    */
      SIMUL_APPR_VAR = SIMUL_APPR_VAR +
                        (((APPR_AVAIL_TIME) / (CYCLE_TIME))**2) ;
      SIMUL_REAL_VAR = SIMUL_REAL_VAR +
                        (((REAL_AVAIL_TIME) / (CYCLE_TIME))**2) ;
      NOW_SIMUL_REAL_AVAIL = SYS_WORKING_STATE / CLOCK ;
      NOW_SIMUL_APPR_AVAIL = (SYS_WORKING_STATE +
                              SYS_FAILURE_STATE) / CLOCK ;
      RUN_WORKING_AVG = RUN_WORKING_AVG +
                        ((SYS_FAILED_TIME + CHK_FAILED_TIME) /
                         CYCLE_TIME) ;
      RUN_FAILED_AVG = RUN_FAILED_AVG +
                       (SYS_DETECT_TIME / CYCLE_TIME) ;
      RUN_DETECTED_AVG = RUN_DETECTED_AVG +
                       (SYS_REPAIR_TIME / CYCLE_TIME) ;
      RUN_CHECKER_AVG = RUN_CHECKER_AVG +
                        (CHK_REPAIR_TIME / CYCLE_TIME) ;
/*                                                                    */
      IF (LOCKING = 5)
        THEN
         DO;
           PREV_SIMUL_REAL_AVAIL = NOW_SIMUL_REAL_AVAIL ;
           PREV_SIMUL_APPR_AVAIL = NOW_SIMUL_APPR_AVAIL ;
         END;
/*                                                                    */
      DIFF_REAL_AVAIL = ABS(PREV_SIMUL_REAL_AVAIL -
                             NOW_SIMUL_REAL_AVAIL) ;
      DIFF_APPR_AVAIL = ABS(PREV_SIMUL_APPR_AVAIL -
                             NOW_SIMUL_APPR_AVAIL) ;
/*                                                                    */
      SYS_FAILURE_VAR = SYS_FAILURE_VAR +
                       ((SYS_DETECT_TIME / CYCLE_TIME)**2) ;
      SYS_DETECT_VAR = SYS_DETECT_VAR +
                      ((SYS_REPAIR_TIME / CYCLE_TIME)**2) ;
      CHK_FAILURE_VAR = CHK_FAILURE_VAR +
                       ((CHK_REPAIR_TIME / CYCLE_TIME)**2) ;
/*                                                                    */
/*  RESET TIME VARIABLES                                              */
/*                                                                    */
      SYS_FAILED_TIME = 0.0 ;
      SYS_DETECT_TIME = 0.0 ;
      SYS_REPAIR_TIME = 0.0 ;
      CHK_FAILED_TIME = 0.0 ;
      CHK_REPAIR_TIME = 0.0 ;
      APPR_AVAIL_TIME = 0.0 ;
      REAL_AVAIL_TIME = 0.0 ;
/*                                                                    */
      EXEC = 0 ;
      SELECT = 1 ;
/*                                                                    */
/*                                                                    */
END 13:                                                               */
```

```
/*                                                                  */
/*  THE FINAL STATISTICS ROUTINE                                    */
/*                                                                  */
      AVG_SYS_WORKING = (SYS_WORKING_STATE / COUNTER) ;
      AVG_SYS_FAILURE = (SYS_FAILURE_STATE / COUNTER) ;
      AVG_CHK_FAILURE = (CHK_FAILURE_STATE / COUNTER) ;
      AVG_SYS_DETECT = (SYS_DETECT_STATE / COUNTER) ;
      AVG_CYCLE_TIME = CLOCK / COUNTER ;
/*                                                                  */
/*                                                                  */
      AVG_CYCLE_REAL = ((AVG_SYS_WORKING) /
                              (AVG_CYCLE_TIME)) ;
      AVG_CYCLE_APPR = (((AVG_SYS_WORKING) +
                              (AVG_SYS_FAILURE)) /
                                                  (AVG_CYCLE_TIME)) ;
/*                                                                  */
      PROB_SYS_WORKING = (SYS_WORKING_STATE / CLOCK) ;
      PROB_SYS_FAILURE = (SYS_FAILURE_STATE / CLOCK) ;
      PROB_SYS_DETECT = (SYS_DETECT_STATE / CLOCK) ;
      PROB_CHK_FAILURE = (CHK_FAILURE_STATE / CLOCK) ;
/*                                                                  */
      VAR_SIMUL_REAL = ABS((SIMUL_REAL_VAR -
                  ((COUNTER)*((AVG_CYCLE_REAL)**2))) /
                                  ((COUNTER) - (1.0))) ;
      VAR_SIMUL_APPR = ABS((SIMUL_APPR_VAR -
                  ((COUNTER)*((AVG_CYCLE_APPR)**2))) /
                                  ((COUNTER) - (1.0))) ;
/*                                                                  */
      VAR_SYS_FAILURE = ABS((SYS_FAILURE_VAR -
                  ((COUNTER)*((AVG_SYS_FAILURE)**2))) /
                        (COUNTER - 1.0)) ;
     VAR_SYS_DETECT = ABS((SYS_DETECT_VAR -
                  ((COUNTER)*((AVG_SYS_DETECT)**2))) /
                        (COUNTER - 1.0)) ;
      VAR CHK_FAILURE = ABS((CHK_FAILURE_VAR -
                  ((COUNTER)*((AVG_CHK_FAILURE)**2))) /
                        (COUNTER - 1.0 )) ;
      VAR_SYS_WORKING = VAR_SIMUL_REAL ;
/*                                                                  */
      FINAL_REAL_DIFF = ABS( NOW_SIMUL_REAL_AVAIL -
                              MODEL_REAL_AVAIL) ;
      FINAL_APPR_DIFF = ABS( NOW_SIMUL_APPR_AVAIL -
                              MODEL_APPR_AVAIL) ;
      STAT = 0 ;
      DUMP = 1 ;
/*                                                                  */
/*                                                                  */
END L4;
/*                                                                  */
```

```
L5: DO WHILE (DUMP = 1) ;
/*                                                                           */
/*   THE 'DUMP' FINAL STATISTICS ROUTINE                                     */
/*                                                                           */
     PUT SKIP(2) LIST('SYSTEM PARAMETERS') ;
     PUT SKIP(2) LIST('  LAMBDA', LAMBDA) ;
     PUT SKIP LIST('  DELTA ', DELTA) ;
     PUT SKIP LIST('  MU    ', MU) ;
     PUT SKIP LIST('  ALPHA ', ALPHA) ;
     PUT SKIP LIST('  BETA  ', BETA) ;
     PUT SKIP(2) LIST('MODEL REAL AVAILABILITY      ',
                     MODEL_REAL_AVAIL) ;
     PUT SKIP LIST('SIMULATION REAL AVAILABILITY',
                NOW_SIMUL_REAL_AVAIL) ;
     PUT SKIP LIST('  DIFFERENCE BETWEEN MODEL & SIMULATE',
                    FINAL_REAL_DIFF) ;
     PUT SKIP LIST('  VARIANCE OF SIMULATION',
                     VAR_SIMUL_REAL) ;
     PUT SKIP(2) LIST('MODEL APPR AVAILABILITY      ,
                     MODEL_APPR_AVAIL) ;
     PUT SKIP LIST('SIMULATION APPR AVAILABILITY',
                NOW_SIMUL_APPR_AVAIL) ;
     PUT SKIP LIST('  DIFFERENCE BETWEEN MODEL & SIMULATE',
                    FINAL_APPR_DIFF) ;
     PUT SKIP LIST('  VARIANCE OF SIMULATION',
                     VAR_SIMUL_APPR) ;
     PUT SKIP(2) LIST('ALL AVERAGES ARE FOR AN AVERAGE CYCLE TIME') ;
     PUT SKIP(2) LIST('AVG CYCLE TIME                          ',
                     AVG_CYCLE_TIME) ;
     PUT SKIP(2) LIST('PROB OF BEING IN SYS WORKING STATE',
                     PROB_SYS_WORKING) ;
     PUT SKIP LIST('  AVG TIME IN SYS WORKING STATE', AVG_SYS_WORKING);
     PUT SKIP LIST('  VARIANCE OF SYS WORKING STATE',
                    VAR_SYS_WORKING) ;
     PUT SKIP(2) LIST('PROB OF BEING IN SYS FAILURE STATE',
                     PROB_SYS_FAILURE) ;
     PUT SKIP LIST('  AVG TIME IN SYS FAILURE STATE', AVG_SYS_FAILURE);
     PUT SKIP LIST('  VARIANCE OF SYS FAILURE STATE',
                     VAR_SYS_FAILURE) ;
     PUT SKIP(2) LIST('PROB OF BEING IN SYS DETECT STATE',
                     PROB_SYS_DETECT) ;
     PUT SKIP LIST('  AVG TIME IN SYS DETECT STATE',
                    AVG_SYS_DETECT) ;
     PUT SKIP LIST('  VARIANCE OF SYS DETECT STATE',
                    VAR_SYS_DETECT) ;
     PUT SKIP(2) LIST('PROB OF BEING IN CHK FAILURE STATE',
                     PROB_CHK_FAILURE) ;
```

```
    PUT SKIP LIST('   AVG TIME IN CHK FAILURE STATE',
                      AVG_CHK_FAILURE) ;
    PUT SKIP LIST('   VARIANCE OF CHK FAILURE STATE',
                      VAR_CHK_FAILURE) ;
    DUMP = 0 ;
    ON = 0 ;
/*                                                                 */
/*                                                                 */
END L5;
/*                                                                 */
/*                                                                 */
/*                                                                 */
/*                                                                 */
          END L0;
/*                                                                 */
/*                                                                 */
/*                                                                 */
/*                                                                 */
 END TRIAL;
```

BIBLIOGRAPHY

1. Anderson, R. T., Reliability Design Handbook, IIT Research Institute, Chicago, 1976, pp. 5-6.

2. Ingle, A. D. and Siewiorek, D. P., "Reliability Models for Multiprocessor Systems With and Without Periodic Maintainence", Department of Computer Science, Carnegie-Mellon University, 1976, p. 3.

3. Hellerman, H. and Conroy, T. F., Computer System Performance, McGraw-Hill, New York, 1975, pp. 84-86.

4. Ramamoorthy, C. V. and Han, Y. W., "Reliability Analysis of Systems with Concurrent Error Detection", IEEE Transactions on Computers, Vol. C-24, No. 9, 1975, pg. 869.

5. Trivedi, K. S., "Mathematical Models for the Design and Analysis of Built-In-Test Equipment", in "Basic Research in Support of Concurrent Fault Monitoring in Modular Digital Systems", Technical Report, Research Triangle Institute, Research Triangle Park, North Carolina, January, 1978.

6. Clary, J. B., "Effectiveness Measures for Built-In-Test Performance Evaluation", Technical Report, Research Triangle Institute, Research Triangle Park, North Carolina, April 1977.

7. Triangle Universities Computation Center, "Document No. LS-110-1", Triangle Research Park, North Carolina.

8. Trivedi, K. S., "Mathematical Models for the Design and Analysis of Built-In-Test Equipment", Department of Computer Science, Duke University, April, 1978, paper to be published.

9. Winkler, R. L. and Hays, W. L., Statistics: Inference, Probability and Decision, Second Edition, Holt, Rinehart and Winston, New York, 1975, pp. 65-69.

APPENDIX B

A HIGH LEVEL DIGITAL COMPUTER SIMULATOR WITH FAULT INJECTION FACILITIES

A HIGH LEVEL DIGITAL COMPUTER SIMULATOR WITH FAULT INJECTION FACILITIES

by

Betty W. Hood

Department of Computer Science
Duke University

Date 5 - 19 - 78

Approved:

P. N. Marinos

Dr. Peter N. Marinos, Supervisor

KS Trivedi

Alan W. Biermann

Thesis submitted in partial fulfillment of the
requirements for the degree of Master of
Arts in the Department of Computer
Science in the Graduate School
of Duke University

1978

ACKNOWLEDGEMENTS

Special thanks to Dr. Peter N. Marinos for his guidance and helpful suggestions throughout the development of the simulator. This research effort was sponsored by the Naval Electronic Systems Command (NAVELEX) through the Research Triangle Institute (RTI) under contract No. N00039-77-C-0363, and the support of Mr. Larry W. Sumney of NAVELEX and Mr. James Clary of RTI is gratefully acknowledged. I also wish to thank Mr. Eric P. Bam for his assistance in formulating Phase-I of the simulator, and Mr. Bruce Ballard for his assistance with the 'C' compiler and use of the 'C' language. To my husband Jerry goes a very special word of thanks for his encouragement, assistance, and support in making this degree possible.

TABLE OF CONTENTS

A HIGH LEVEL DIGITAL COMPUTER SIMULATOR WITH FAULT INJECTION FACILITIES

1. INTRODUCTION

A major consideration in the design and development of a computer system today is the reliability of that system. The need for high reliability in aerospace computers and computers used in military applications is obvious. This need is expanding to encompass many other areas such as banking, stock exchanges, and communications, and will doubtless continue to be important as the range of uses of computer systems increases.

The National Aeronautics and Space Administration defines reliability as "the probability of a device performing adequately for the period of time intended under the operating conditions encountered." Two major approaches to the study of reliability today are (1) the analytical model, and (2) the consideration of certain real hardware organizations such as the Jet Propulsion Laboratories STAR (Self-Testing and Repairing) computer.

The analytical model requires the analytical description of modules and thus the system collectively. Statistical distributions are then used to approximate the occurrence of faults and fault-tolerant processes (detection, correction, etc.). This approach allows a wide range of system configurations to be examined. However, the reduction of a real world implementation to a series of mathematical equations immediately becomes nontrivial and very soon becomes intractable (8).

Real hardware organizations, unlike analytical models, facilitate performance evaluation based on the processing of typical workloads given. This is an evaluation relatively simple compared to that of analytical models. However, actual hardware configurations do not allow the functional flexibility required for the study of a wide range of system configurations, and an alternative approach should then be considered.

Ideally, a system simulator capable of supporting a wide range of system configurations in addition to allowing the use of statistical distributions to approximate various fault environments is needed. This of course must be accomplished with minimal cost. The solution proposed here is a simulator which combines the functional flexibility of a simulated hardware organization with the ability to process a typical workload on the simulated machine subject to a specified fault environment. The simulator is capable of supporting many system configurations in addition to providing fault-injection facilities.

2. OBJECTIVES

There were many considerations in the development of the simulator. As discussed earlier, a large number of system configurations should be supported to allow the user flexibility. The simulator should also support a wide range of functions, in particular, functions which would allow the simulation of faults occurring in real systems. The ease of use was also a major consideration.

More specifically, the simulator proposed here allows the user to accomplish the following:

(1) to configure the system(s) under investigation,

(2) to generate a fault environment and distribute faults over a specified mission time using appropriate statistical distributions,

(3) to program fault-detection capabilities to study manifestations of faults,

(4) to program fault-correction capabilities, and

(5) to observe the performance of system(s) with regard to a typical workload when such systems are subject to various fault environments.

Traditionally, simulators have been somewhat restrictive in the support of a wide range of systems. This simulator resolves this problem by providing basic system building blocks which allow the user to create a system. Also provided are special fault-processing instructions which allow the injection of faults at will in the created system. Additionally, an instruction set is provided which allows the user to program the system to detect and recover from injected faults in a prescribed manner.

The user may devise a typical workload suitable for processing on the simulated system and note the performance of the system subject to various fault environments. In general, parameters may be inserted into the system accounting for the system configuration, the fault environment, the distribution of faults over time, and the fault-detection and fault-correction algorithms. To accomplish the above with ease, instructions provided are similar to those of a simple assembly language. The user is, therefore, required to spend a minimal amount of time learning to program the system.

3. SIMULATOR DESCRIPTION

A fault may be defined as "any change in a system which causes it to behave differently from the original system" (3). These may be classified as logical or parametric. A logical fault is one which causes the logic fuction of a circuit element (or elements) of an input signal to be changed to some other signal (3). Examples of logical faults are stuck-at-one (s-a-1) faults where a circuit signal becomes stuck at the logical value of 1, and stuck-at-zero (s-a-0) faults where a circuit signal becomes stuck at the logical value 0. Parametric faults frequently alter the magnitude of a circuit parameter causing a change in some factor such as circuit speed, current, or voltage (3). This simulator is concerned with the simulation of logical faults.

Logical faults may be further classified as permanent and transient. The stuck-at faults mentioned above are examples of permanent faults, i.e. once the fault occurs it permanently becomes a part of the system unless corrected. A transient fault exists over a given time after which the faulty signal reverts to the fault-free mode. These may have many causes, for example, a change in environment (temperature, humidity), or a change in system load. Transient faults have been found to be responsible for up to 85% of faults in systems today (10).

3.1 Development

Throughout the development of the simulator the concepts of wide applicability, functional capability, and ease of use were emphasized.

With these considerations in mind, the first design was proposed. The user was allowed total freedom in creating a system by defining basic units and then specifying a configuration of these units into a system. Unit specifications included field sizes (i.e., number of bits), field types, (i.e. whether the fields were input fields or output fields or both), the number of fields within the units, and the means of transmission (serially or in parallel) into and out of the fields of the unit.

Inherent in this design were many problems for both the user and the implementor. Because the units were defined at a very low level, many specifications were required that had not been considered initially. If a unit contained more than one input field and one output field, it was necessary to specify how the fields within a unit were connected, i.e., how fields were connected to each other. These specifications also presented inconveniences in the programming of the created system in that the user was required to specify many operations in order to perform very simple tasks.

Additionally, the implementation of this scheme posed many problems. It was very difficult to create a system which entailed a variable number of unit types, a variable number of units within type, a variable number of fields within units, three possible types of unit fields (input, output, or both), and also a variable number of bits within fields. Specifications for all interconnections between fields within units and between fields of different units were also required. This was very cumbersome.

Many checks were also necessary when data was moved between fields. It was necessary to check whether the fields were of the same size, whether data transmission between fields was compatible (serially or in parallel), and whether data was being moved illegally, e.g., data was being transmitted from an input field of one unit into an output field of another unit.

The generation of understandable object code was also nearly impossible. Since word sizes in the units (including memory) were variable, the formatting of object code to be loaded into memory was difficult, and any sort of addressing scheme became unnecessarily complex to implement.

Thus, for the benefit of both the user and the implementor, a new design approach was considered.

In the scheme which was finally adopted, instead of the user defining units, a resource pool with six basic unit types is provided from which a system may be configured. These are arithmetic logic units (ALU's), central processor units (CPU's), memory units consisting of up to 4096 words (2 ** 12 words), bus units, peripheral device units, and voter switch discriminator (VSD) units. All registers within the units are of a fixed length of thirty-two bits compared to the variable number of bits in the previous design. A system can be configured using these basic units with up to a total of 128 units (provided there are no memory space limitations in the host machine). For further documentation see Section 3.2.

The progamming language which was needed to perform operations on the created system was developed with the following considerations. In order to facilitate the injection of faults at the bit level, a low-level language was considered desirable. It was also necessary that the language support the modeling of typical logical faults. Additionally, it was important that the user be able to learn the language with ease and with minimal investment of time and effort.

The language is based on an assembly-type language with special instructions added to facilitate the modeling of specific faults. Since the language is very similar to an assembler language, the programmer should have no difficulty learning its use quickly. For further documentation see Section 3.3.

Thus, the simulator proposed consists of three major phases: Phase-I, the definition phase, Phase-II, the cross-assembler phase, and Phase-III, the execution phase. Like most simulators today, it is an event-driven simulator rather than a compiler-driven simulator. An event in this case is defined as a change in value of a signal line.

3.2 PHASE-I: Definition Phase

The objective of the definition phase of the simulator is to obtain a description of a digital system from the user and generate appropriate data structures to be used by Phase-II and Phase-III of the program. As discussed previously, the user is given a set of functionally pre-defined units with which the system must be constructed (i.e. arithmetic logic units, central processing units, memories, busses, peripheral devices, and voter switch discriminator units or VSD units). Each of these units has been assigned appropriate input, output, and i/o registers which are classified as follows:

TYPE	LABEL
1) data register	d, d1, d2, etc.

2) mode register	m
3) status register	s
4) address register	a
5) general purpose registers	r1, r2, r3, etc.

As a note of clarification, these register-types can serve in either an input, output, or i/o capacity with respect to a particular unit-type.

3.2.1 Unit structures

The basic units are constructed as follows:

Arithmetic Logic Unit

FIELD	SYMBOL
general purpose register 1	r1
general purpose register 2	r2
mode register	m
data register	d
status register	s

General purpose registers one and two are used as the two input registers, and the data register is used as the output register.

Memory Unit

FIELD	SYMBOL
address register	a
status register	s
data register	d

The address register may hold the address of the word to be fetched from memory, with the data register holding the fetched word.

Central Processor Unit

FIELD	SYMBOL
mode register	m
status register	s
general purpose register 1	r1
general purpose register 2	r2
general purpose register 3	r3
general purpose register 4	r4
general purpose register 5	r5
general purpose register 6	r6
general purpose register 7	r7
general purpose register 8	r8

Bus unit

FIELD	SYMBOL
status register	s
data register	d
address register	a

Voter Switch Discriminator Unit

FIELD	SYMBOL
mode register	m
data register 1	d1
data register 2	d2
data register 3	d3
data register 4	d4
data register 5	d5
data register 6	d6
data register 7	d7
data register 8	d8
status register	s
data register	d
general purpose register 1	r1

Peripherals

FIELD	SYMBOL
mode register	m
data register 1	d1
status register	s
data register	d

The status registers in each of the units indicate whether the unit is operational or non-operational, and can be set or reset by the user.

The user may utilize as many or as few of the fields within the units as desired. The units are constructed to serve in a very general purpose manner in order to allow the simulation of as many systems as possible.

3.2.2 System Creation

The task of defining a digital system is now one of creating the units and interconnecting the unit registers or fields of the units. A high level, symbolic language has been developed which enables the user to describe these interconnections to the simulator.

The creation of a unit is accomplished by specifying the general form "unit-identifier-name.fieldname", where the first character of the field name indicates the unit type ("a" for alu, "m" for memory, and so forth), and the remainder of the field is the symbol of one of the designated fields of the unit (e.g. d1, r1, s). Thus "Alu1.ar1" creates an arithmetic logic unit with the identifying name "Alu1".

Field connections are specified easily by using ">" between two field names. This indicates that the contents of the field on the left of the ">" may be transmitted to the field on the right of the ">". For example, "Alu1.ad >

Vsd1.vr1" creates two units, an arithmetic logic unit with identifying name "Alu1" and a voter switch discriminator unit with identifying name "Vsd1". The statement also specifies that the contents of the data register of "Alu1" may be transmitted to general purpose register 1 of "Vsd1". Thus, an entire system may be created using similar specifications. (Formal language constructs of the above may be found in Section 4.1)

3.2.3 Program

The definition phase of the simulator is an interpreter whose inputs are the command lines described above and whose outputs are a series of tables which describe the defined system. This language was developed so that, in a single pass, the input file is scanned character by character with no look-ahead or look-behind required.

Every occurrence of a properly constructed identifier in the input stream causes the interpreter to search for the label portion of that identifier to determine if a similar label has been used previously. If the identifier has appeared before, the unit-types of the new identifier and the old one are checked for consistency. Otherwise a new unit tagged with the specified label is allotted space. Thus, the appearance of an identifier in the input stream causes the associated unit to be defined or checked for consistency. This completely eliminates the need for declaration statements in the language.

The interpreter is also characterized by powerful error-detecting capabilities. Because only a single pass is utilized and because of the scanning techniques employed, the exact location of an error is always known. In the event of an error, an appropriate error message is printed and the exact location of the infraction is indicated. In order to prevent confusing or unrelated error messages, the remainder of the command line is then ignored.

3.3 PHASE-II: Cross-Assembler

One of the unique features of the simulator is its fault-injection capability. This capability is incorporated in the programming language designed to program the system created in Phase-I. As was mentioned previously, this language is similar to an assembly language but with additional instructions added to facilitate the modeling of typical faults. This language will henceforth be called the assembly language or assembler. The objective of the cross-assembler is to read the user's assembly program, decode the instructions, generate the object deck, and load it into a specified memory.

3.3.1 Develcpment of Instruction Repertoire

The major consideration in the development of the programming language was the devising of an instruction repertoire which wculd support the modeling of as many logical faults as possible. This included both permanent and transient faults. Ease of use was also a significant consideration.

Since the occurrence of faults is manifested at the register level, in order to model these faults it was important that the user be able to easily access the registers. A register-level language was thus considered desirable. This immediately indicated an assembly-type language.

An assembly-type language very nicely supported the injection of transient faults provided the faults were of short duration. The modeling of permanent faults and transient faults cf a very long duration however was inconvenient. The user was required tc repeat the injection of a transient fault many times in order to accomplish these effects.

Instructions were then devised to facilitate the modeling cf permanent faults. Thus, transient faults of short duration and permanent faults were accommodated. To model transient faults of longer duration, additional instructions were added to allow the alteration of permanent fault-injection instructions executed previously. Thus, assembly type instructions enhanced by special fault-injection instructions accomplished the modeling of the desired faults to be simulated.

One additional feature that was desirable to incorporate in the simulator was some means of indicating a time when certain faults could be injected. In the earlier discussion, it was assumed that the user would write a typical workload program and inject faults into fields that were previously referenced in the program. Since such programs could get rather lengthy, it was unreasonable to expect the user to calculate the position in a program where certain faults should be injected by manually working through the timing of the program instructions. Thus, an additional "timing" feature was implemented where the user may specify the times specified faults are to occur. This feature is extremely useful in modeling specific distribution times of faults (e.g. Poisson).

3.3.2 Discussion of Instructions

There are two basic instruction types with which to program the system created in Phase-I, "regular" instructions and "fault-injection" instructions. Within the fault-injection instructions are two classes, (1) instructions nearly identical to the "regular" instructions, and (2) special fault-injection instructions. The basic form for all instructions of the assembler is as follows:

label: OP operands

A label (optional) must be less than or equal to eight alphanumeric characters in length and must be immediately followed by a colon. "OP" must be a valid instruction mnemonic with the operands being interpreted according to the specific instruction. Pseudo operations are also supported and are specified by a "$" as the first character. Formal presentation of all assembler instructions is found in Section 4.2.

The "regular" instructions are essentially equivalent to an assembly language. There are six basic instruction types: (1) two-operand (2) one-operand (3) branch (4) immediate-operand (5) unit-operand and (6) other. These are very similar to typical assembly language instructions with the exception of the unit operand instructions. Brief discussion of these instructions follows.

The two-operand instructions are those which expect two field names or a field name and a label as operands. As an example "AD Cpu1.cr1,Cpu1.cr2" will add the contents of registers one and two of Cpu1 and place the result in register one of the CPU. An "I" (preceded by a comma) may follow the second operand to indicate that indirect addressing is to be used in obtaining the value for the second operand. The other two-operand instructions adhere to the same format with the exception of the compare instruction (C) and the memory reference instructions (LM and STM). The compare instruction compares the two operand values and sets the condition code register accordingly. The load register from memory and store from register to memory instructions expect the second operand to be a label instead of a field name.

The one-operand instructions expect a single field name as an operand and the field is changed according to the instruction specified. For example, "NOT Cpu1.cr1" generates the ones complement of the original field value in Cpu1.cr1.

The immediate-operand instructions expect two operands, the first operand being a field and the second operand being an integer value. As an example, "SBI Cpu1.cr1,39" subtracts the decimal value 39 from the contents of Cpu1.cr1. The other immediate operand instructions adhere to this format. The compare instruction again sets the condition code register, and the shift instructions shift the specified field value (right or left as indicated by the instruction) according to the number specified in the immediate value.

The branch instructions include all branch instructions plus the "RET" (return) and "HLT" (halt) instructions. Conditional branches have a condition code number associated with them which is compared to the condition code register. If any of the set bits match, the designated branch is executed. The condition codes are indicated as follows (leftmost bit is bit 0):

Condition	Bit number set	Condition Code Number
equal	31	01
less than	30	02
greater than	29	04
parity (odd)	28	08
parity (even)	27	016

To branch on more than one condition, the sum of the corresponding condition code numbers will generate the proper condition. For example, "BC 04,label1" will branch provided the "greater than" bit is set. To branch on the condition "greater than or equal to" the instruction should appear as "BC 05,label1". It should be noted that the condition codes are not checked by the simulator for rational usage; therefore, unconditional branches using conditional branch instructions are possible. For example, "BC 07,label1" branches on the condition "less than, equal to, or greater than", i.e. branches unconditionally. Such condition codes should be avoided. The branch and save (BSA) instruction allows branching to subroutines by storing the program counter word on a stack. The return (RET) pops the stack and restores the program counter to the word following the address popped. The "HLT" instruction effectively stops the program by branching to the end of the program.

The unit operand instructions were added to facilitate the user in performing certain operations involving an entire unit. These instructions differ from each of the previous instructions in that a unit name instead of a field name or label is specified as an operand. Additionally, the various instructions expect the unit operands to be of a particular type. For all of the arithmetic unit operations, the unit type must be an ALU. The instruction triggers the ALU in that the two inputs are operated on as specified, the result is placed in the output register and the output directed as specified by the connections given in Phase-I. The clear unit (CLU), dump unit (UDUMP), and SST and RST (set and reset status) will accept the name of a unit of any unit type as an operand and perform the necessary operations. The memory dump (MDUMP) requires the name of a memory unit as an operand and prints the contents of the locations specified by the addresses given as additional operands. The VOTE instruction expects the name of a voter-switch-discriminator (VSD) unit as an operand and triggers VSD units in the same manner that unit arithmetic instructions trigger ALU's.

The "other" instructions include any instructions not covered by the above, e.g. input/output. The format and use is clearly specified in Section 4.2.

This concludes the presentation of the "regular" instruction types. The similarity to an assembly language is thus obvious. The following discussion details the fault-injection instructions.

As was mentioned above, there are two classes of the fault-injection instructions (1) those similar to the "regular" instructions and (2) the special fault-injection instructions. Fault-injection instructions are immediately differentiated from regular instructions by the first character of the instruction mnemonic. The first character of every fault-injection instruction is an asterisk (*) compared to a letter of the alphabet for regular instructions.

The fault-injection instructions which are similar to the regular instructions are characterized by identical instruction mnemonics with the addition of the asterisk as the first character (e.g. *AD). The difference between these fault-injection instructions and their "regular" counterparts lies in the amount of time required to perform the operations. As is well known, the decoding of an instruction and the execution of the operation specified require a certain amount of clock time within the computer. This time span is simulated for the regular instructions. However, this time span is <u>not</u> simulated for the fault-injection instructions which are effectively decoded and executed in zero time. This allows the user to effectively "stop the clock" and alter any of the values in the units as desired by using the necessary fault-injection instructions. For instance, the user may specify "*ORI Alu1.ar1,04" which will effectively force bit 29 of Alu1.ar1 to the logical value 1, i.e. a stuck-at-1. Any of the "regular" instruction operations may thus be performed as fault-injection instructions in the above manner with no time added to the clock. Hence, these instructions may be used to inject certain faults.

The above feature allows the user to inject transient faults of a short duration. It should be noted that these injected faults will be sustained until the field is accessed by another operation which changes its value.

To facilitate the simulation of permanent faults, the special fault-injection instructions "SA0" (stuck-at-0) and "SA1" (stuck-at-1) were developed. For example, to permanently fix bit 29 of Alu1.ar1 to the logical value 1, the instruction "SA1 Alu1.ar1,29" may be used. With every reference of the field, bit 29 will be fixed at the logical value 1. The instructions allow any number of bits to be specified to be stuck-at a logical value, for example, "SA0 Alu1.ar1,3,15,22" fixes bits 3, 15, and 22 to the logical value 0. Later, "SA0 Alu1.ar1,25" would fix bit 25 to logical 0 in addition to the previous bits stuck-at logical 0.

In order to simulate transient faults of a longer duration than that provided by the assembly-type fault-injection instructions, additional special fault-injection instructions were developed which allow the user to "unstick" previously stuck-at bits. These were the "remove stuck-at-0" (*RF0) and the "remove stuck-at-1" (*RF1) instructions. As an example, the instruction "RF0 Alu1.ar1,15" would "unstick" bit 15 which was stuck-at-0 in the previous paragraph, i.e. bit 15 would no longer be stuck-at-0 but would hold the actual logical value assigned by an operation.

To enhance the stuck-at instructions, the "random stuck-at" fault-injection instructions were devised. These allow stuck-at faults to be injected at a certain frequency specified in the instruction. For example, "RSA0 Alu1.ar1,13,19;1/77" states that "one of every 77 times that Alu1.ar1 is referenced, fix bits 13 and 19 to the logical value 0". A random number generator is used to return a number which is checked against the fraction specified. If the number returned is less than or equal to the fraction, the bits are fixed to the logical value specified. Otherwise, no fault-injection is performed. Thus, bits may be randomly stuck-at logical values.

With extensive use of these instructions, the "bookkeeping" required of the user in order to know the current stuck-at values of each field could become very cumbersome. To mitigate this situation, the "remove all faults" (RAF) instruction was thus devised to clear all faults from a field and allow the user to start the injection of faults anew.

A final special fault-injection instruction unrelated to those above is the *DE (dead-end) instruction. This allows the user to increase or decrease the execution operation times of instructions. For example, the sequence

```
ADI   Cpu1.cr1,52
*DE   Cpu1.cr1, 2298
```

states "add 2298 time units to the original execution time" for the field Cpu1.cr1. Effectively then, the addition operation will require the normal addition operation time plus an additional 2298 time units. This feature allows operation times to be easily altered dynamically, thus accounting for possible degradation due to system faults. (Operation times may be also changed by changing specifications in the DEFINE file--see Section 3.5).

As was mentioned in Section 3.3.1, the user at this time is provided with the capabilities of injecting faults but has no means of specifying when those faults are to be injected without strenous computation. To aid the user in this respect, a separate file may be specified which contains fault-injection instructions of a special format which support a timing feature. The following example illustrates this feature:

```
     526     *SA0     Alu1.ar1,20, 14, 2,19
     874     *RSA1    Alu2.ar2, 17, 31,22
     259     *OEI     Cpu1.cr1, 16
    3586     *RAP     Alu1.ar1
```

The numbers on the left of the fault-injection instructions indicate the times the particular faults are to be injected. The user may thus calculate fault times over a given mission time using a statistical distribution and insert the faults according to this distribution. It is very important to note however that this is the only file where times may be assigned, and that no labels or regular instructions should appear in this file. The program will not be executed if the format specified is not adhered to.

3.3.3 Program

The program in Phase-II is very much like an assembler with added features to accommodate the unique facilities of the simulator. It consists of four basic parts: (1) an input section (2) a section to decode and queue the fault-injection time initializations (3) a section analogous to pass-one of an assembler and (4) a section analogous to pass-two of an assembler.

The first section of the assembler reads input indicating the memory where the program will be loaded. A check is made of its validity and the beginning address of the specified memory calculated. The name of the file containing the Phase-II program is then also read and checked.

The second section initially checks for the presence of the special fault initialization file. If not present, the first section is complete as no processing is required. If it is present, the times specified are read, the instructions decoded, and the specified events placed on the event queue at the designated times. Further discussion of the placement on the event queue is found in Section 3.4.

The bulk of the work of Phase-II is accomplished in the third section. In this section, the instructions and pseudo operations are decoded, the labels and their corresponding addresses are stored in a symbol table, and all of the binary code for all of the instructions except branch instructions and memory reference instructions (which require labels) is generated. The addresses of the memory words corresponding to the branch and load and store from memory instructions are also stored in a table for use in the third section. Thus, all binary code except for the instructions with labels as operands is completed in this section.

The fourth section completes the encoding by inserting the label address for the instructions with labels as operands. Thus, at the end of this section, the binary code is loaded into memory and ready for execution in Phase-III of the simulator.

Throughout Phase-II, appropriate syntax checks of the user's program are made. In the event of an error, an error message is produced and an error count is incremented. If the error count is greater than zero at the end of Phase-II, no execution is attempted and the user is urged to correct the errors indicated and run the program again.

3.4 PHASE III: Execution

The purpose of Phase-III of the simulator is to perform the actual execution of the instructions from Phase-II of the simulator on the system created in Phase-I. This requires that the binary code which was loaded in memory be deciphered and the proper actions taken.

3.4.1 Description

Of fundamental importance in the simulation is the proper interpretation of the instructions and the correct scheduling of the specified operations. This is accomplished by (1) creating and maintaining an event queue and (2) maintaining a record of the last completion time associated with a field. Both of these actions are essential parts of the simulation.

An integral part of this scheme is the presence of a universal clock. The decoding, scheduling, and performance of all operations inherently depend on the clock.

To illustrate the need and use of the above and also to illustrate how Phase-III operates, a sample program segment is given in Figure 3.4.1. The code in the example calculates the quantity ((25*35) + 2**20) and places the result in register 1 of Cpu1. The scheduling and execution of this program segment are presented in Figure 3.4.3 using the operation times specified in Figure 3.4.2.

The execution of the load instructions is very straightforward. The execution of the multiplication requires operands found in two registers. However, at the time the instruction is decoded, one of these registers (Cpu1.cr2) is being altered by a previous instruction. The status of the registers must be "remembered" by the simulator and is done so by means of keeping a "last completion time" record for each field in the simulator. Thus, the multiplication may begin at the time both fields have completed previous operations, i.e. clock value 10. Since a multiplication requires 19 time units, the new completion time for the fields is now 29. The execution of the addition instruction and the shift instruction are handled in a similar manner. The last completion record is also needed for the proper scheduling of the fault-injection instructions as is evidenced by instruction (5). The stuck-at-0 fault is to be injected after the logical shift has been executed, thus, the fault should be injected at the time the logical shift is completed, i.e., clock value 15.

The need for an event queue to schedule events properly and a completion time record to prevent overlapping operations and fault scheduling should be evident. The importance of the clock is also illustrated.

Another integral part of the simulator is the injection of the stuck-at faults. Each field of the simulator has a fault register associated with it (transparent to the user) which indicates which (if any) stuck-at faults are to be injected. To facilitate the injection of these faults, a stuck-at-1 mask, a stuck-at-0 mask, a random-stuck-at-1 mask, and a random-stuck-at-0 mask are also associated with each field. Additionally, the random stuck-at masks require an associated frequency field. To inject the stuck-at-0 faults, a logical AND of the field value and the mask is performed with the "faulted" field value as a result. To inject the stuck-at-1 faults, a logical OR is performed.

Instruction Number	Instruction
(1)	LI Cpu1,cr1,25 (load reg. with value 25)
(2)	LI Cpu1.cr2,35 (load reg. with value 35)
(3)	LI Cpu1.cr3,1 (load reg. with value 1)
(4)	M Cpu1.cr1,Cpu1.cr2 (multiply registers)
(5)	SLL Cpu1.cr3,20 (shift left logical 20 bits)
(6)	*SAO Cpu1.cr3, 13,19 (s-a-0 bits 13 and 19)
(7)	AD Cpu1.cr1,Cpu1.cr3 (add registers)

Figure 3.4.1: Sample program segment.

Operation	Operation time
Fetch and Decode	2 time units
Logical shift	3 time units
Load	6 time units
Addition	10 time units
Multiplication	19 time units

Figure 3.4.2: Sample operation times (arbitrary).

Clock value	Operations performed
0	
2	Instruction (1) decoded, placed on the event queue, completion time = 8
4	Instruction (2) decoded, placed on the event queue, completion time = 10
6	Instruction (3) decoded, placed on the event queue, completion time = 12
8	Instruction (1) execution completed Instruction (4) decoded, placed on the event queue, completion time = 29
10	Instruction (2) execution completed Instruction (5) decoded, placed on the event queue, completion time = 15
12	Instruction (3) execution completed Instruction (6) decoded, placed on the event queue, completion time = 15 Instruction (7) decoded, placed on the event queue, completion time = 39
15	Instruction (5) execution completed Instruction (6) execution completed
29	Instruction (4) execution completed
39	Instruction (7) execution completed

Figure 3.4.3: Scheduling and execution of program segment in Figure 3.4.1 using operation times in Figure 3.4.2.

3.4.2 Program

The following sequence summarizes the operations performed in Phase-III of the simulator:

```
            Initialize clock
    LOOP until all instructions are processed:
            Execute instructions with completion
               times <= clock
            Decode and queue instruction
            Increment clock by DECODETIME (if
               not fault-injection)
    End LOOP.
```

The clock is initialized to zero, and all instructions to be executed at that time (any fault initializations specified at time 0) are performed. The first instruction is then decoded and placed on the event queue, and the clock is incremented if the instruction was a regular instruction. The execute, decode and queue, and increment sequence is then repeated until all instructions are processed.

The event queue is represented by a doubly linked list, with each node representing an operation specified by an instruction in Phase-II. The entries on the list are sorted in ascending order according to the event completion time. Thus, the execution routine removes events from the queue and performs the specified operation until the event completion time of a node on the queue exceeds the clock value. The next instruction is then decoded and placed on the queue and the clock value incremented if necessary.

3.5 Usage

The user is expected to supply a program for both Phase-I and Phase-II of the simulator. These programs should conform to the syntax indicated in Section 4.1 for Phase-I (creating the system) and Section 4.2 (programming the system).

When the object file for the simulator is executed, the user will be requested to supply the following information:

(1) the name of the file containing the program for Phase-I,

(2) whether the fault initialization file for Phase-II is present and the name of the file if it is present

(3) the name of a memory created in Phase-I where the program from Phase-II will be loaded, and

(4) the name of the file containing the program for Phase-II

The desired system will then be simulated.

To support the simulation of as many machines as possible, a DEFINE file is provided which allows operation times to be changed at the user's will. Operations are listed and their corresponding operation times listed next to them. For example, "define ADDTIME 8" states that the operation time for an addition operation is eight clock units. To change this, the user need only change the "8" to the desired value. The define file is included within the simulator and the operation times given in the DEFINE file are the ones used in the simulation.

3.6 Simulator Flowcharts/Outlines

In order to aid the understanding of the program, the following flowcharts/outlines are given. The following conventions are used:

(1) All subroutine names are capitalized in order to distinguish them from other information (e.g. CODEINST).

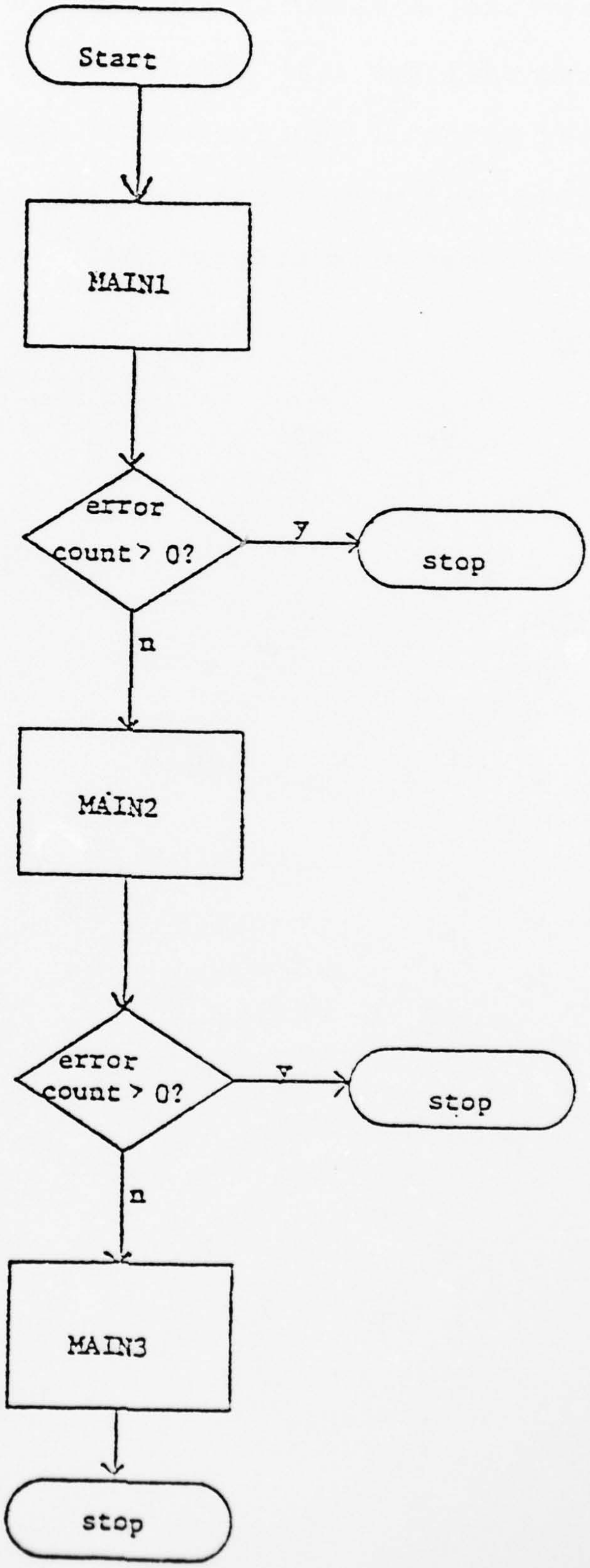
(2) The increment convention of the language 'C' is used (e.g. "progaddr =+ 2" indicates that the variable progaddr is incremented by 2).

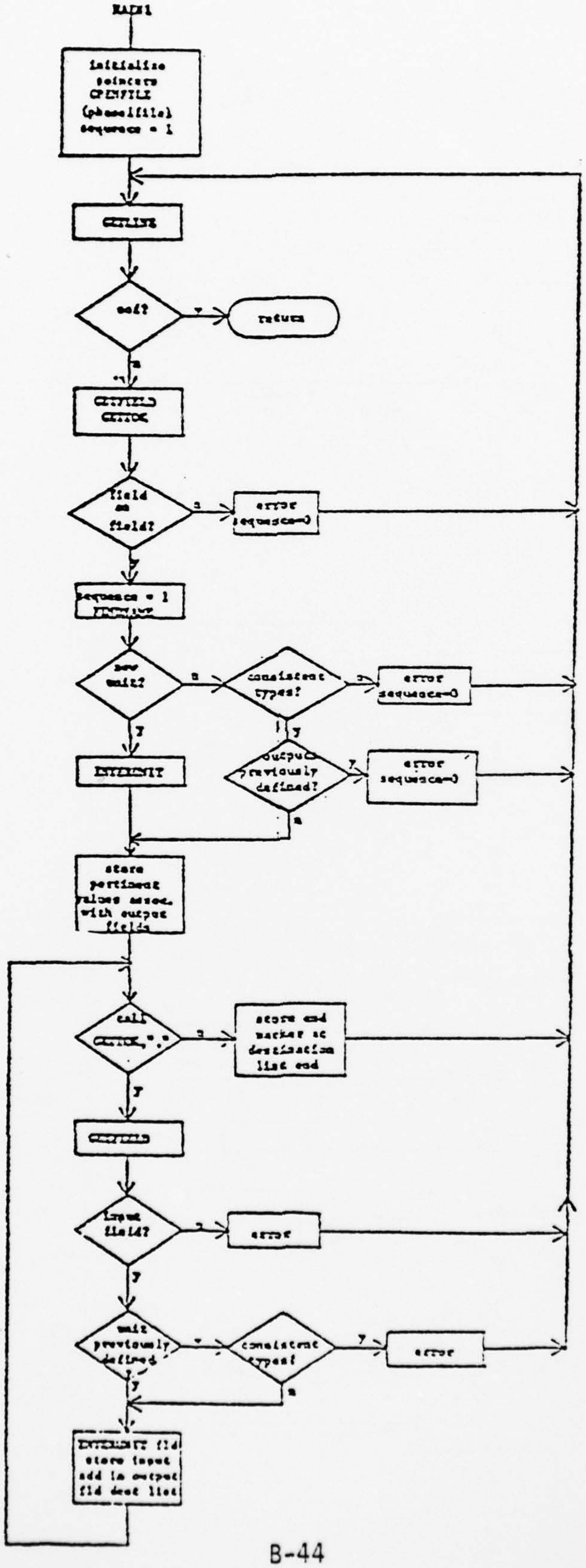
(3) If the purpose of the subroutine is

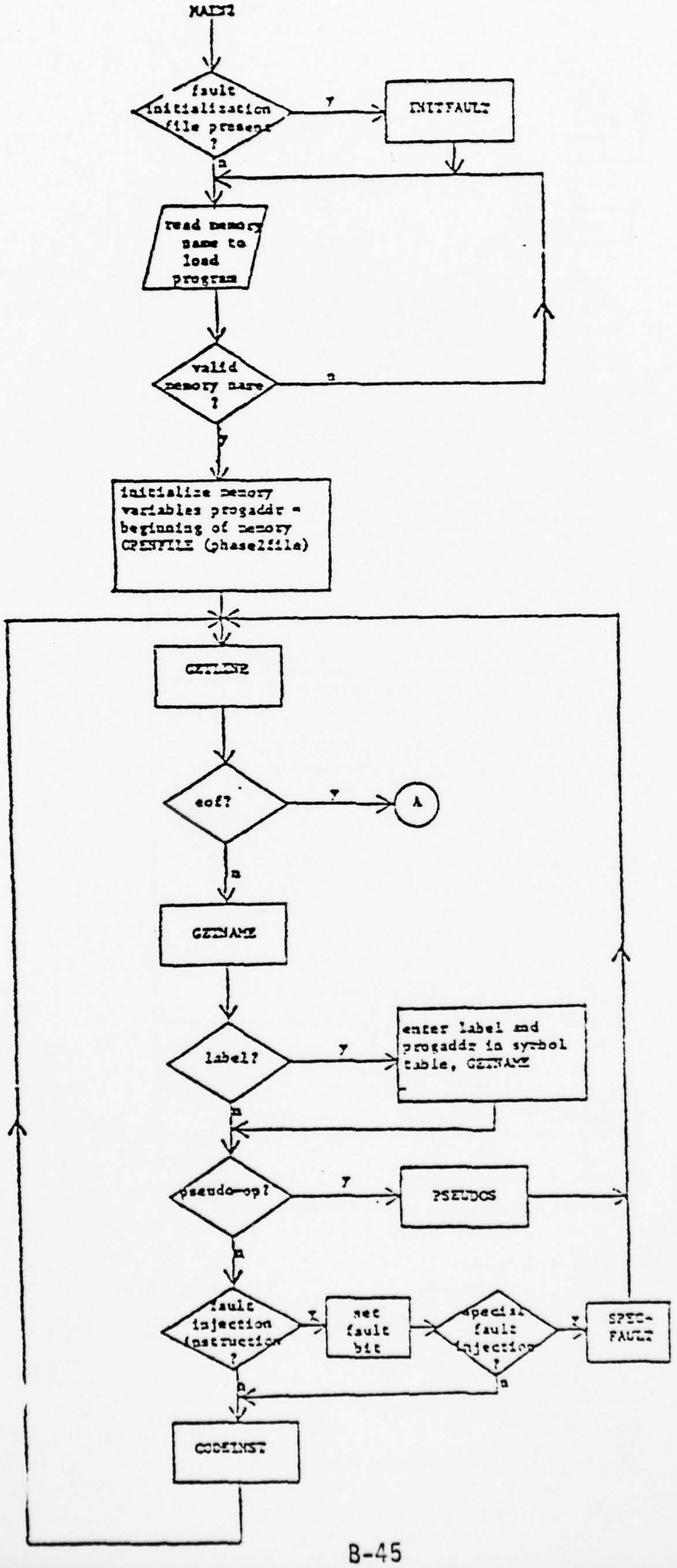
obvious, a flowchart/outline is not included for that routine (e.g. GETLINE).

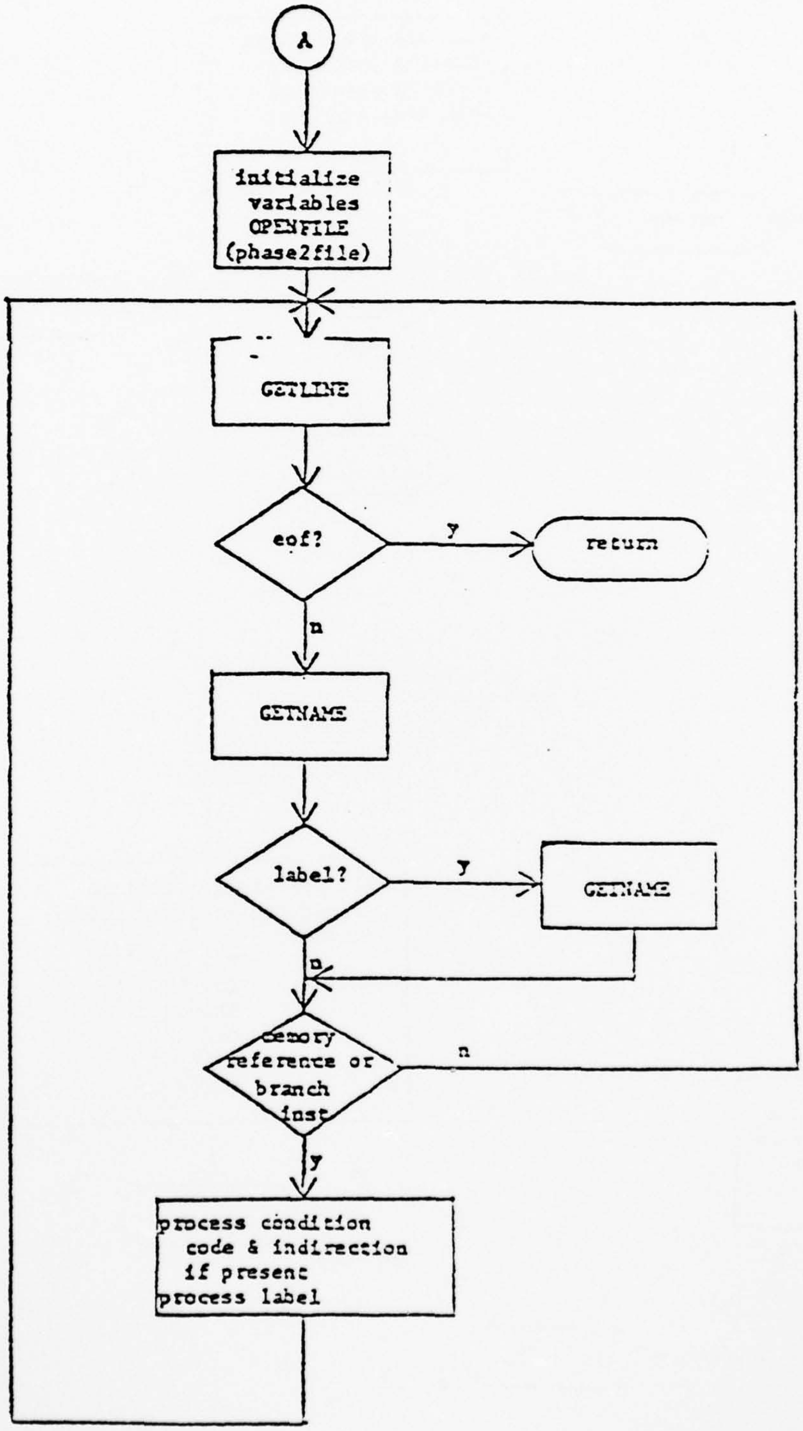
Instead of using strictly a flowchart format, many of the processing boxes include brief outlines as it was felt that this would aid the reader's understanding.

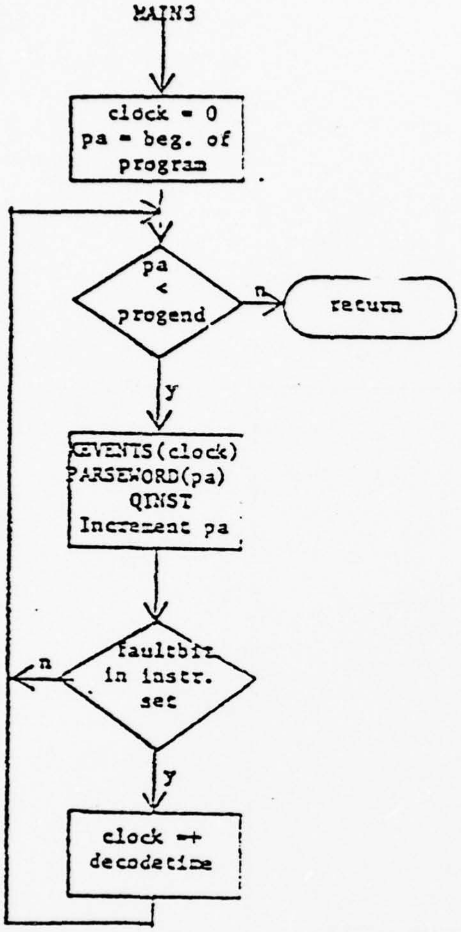
MAINLINE

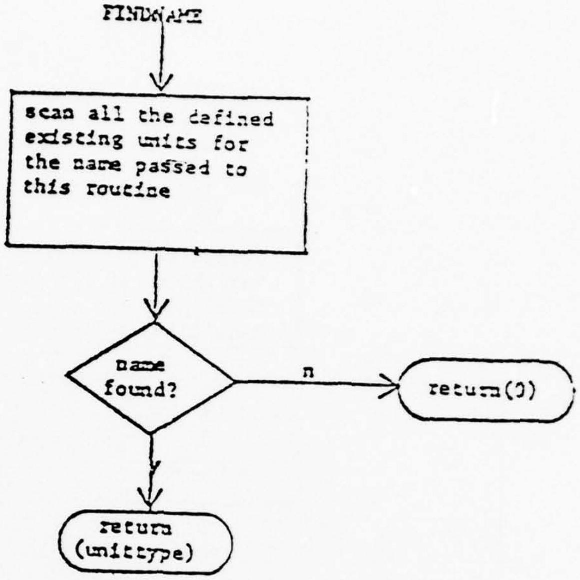

Start
MAIN1
error count > 0?
y
stop
n
MAIN2
error count > 0?
y
stop
n
MAIN3
stop

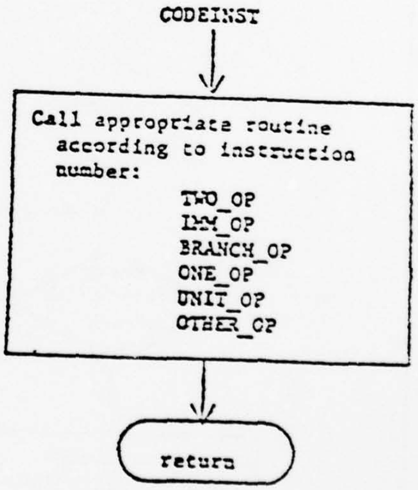

MAIN1
initialize pointers OPENFILE (phase1file) sequence = 1
GETLINE
end?
y
return
n
GETFIELD GETTOK
field an field?
n
error sequence=0
y
sequence = 1 previous
new unit?
n
consistent types?
n
error sequence=0
y
output previously defined?
y
error sequence=0
n
y
ENTERUNIT
store pertinent values assoc. with output fields
call GETTOK,"="
n
store end marker at destination list end
y
GETFIELD
input field?
n
error
y
unit previously defined
y
consistent types?
y
error
n
y
ENTERUNIT fld store input add in output fld dest list

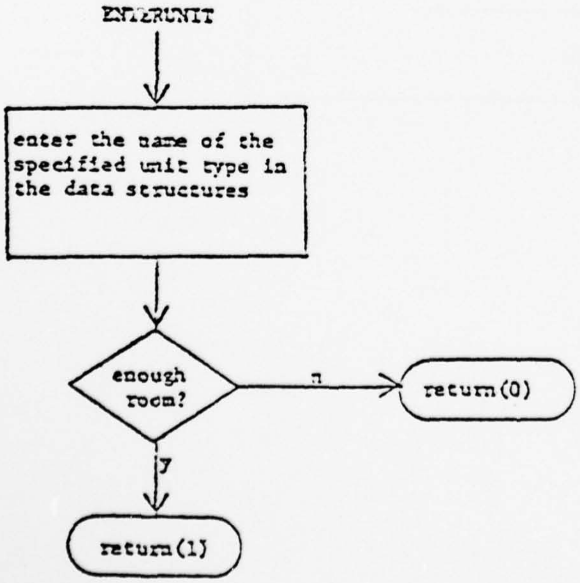

MAIN2
fault initialization file present ?
y
INITFAULT
n
read memory name to load program
valid memory name ?
n
y
initialize memory variables progaddr = beginning of memory OPENFILE (phase2file)
GETLINE
eof?
y
A
n
GETNAME
label?
y
enter label and progaddr in symbol table, GETNAME
n
pseudo-op?
y
PSEUDOS
n
fault injection instruction ?
y
set fault bit
special fault injection ?
y
SPEC-FAULT
n
n
CODEINST

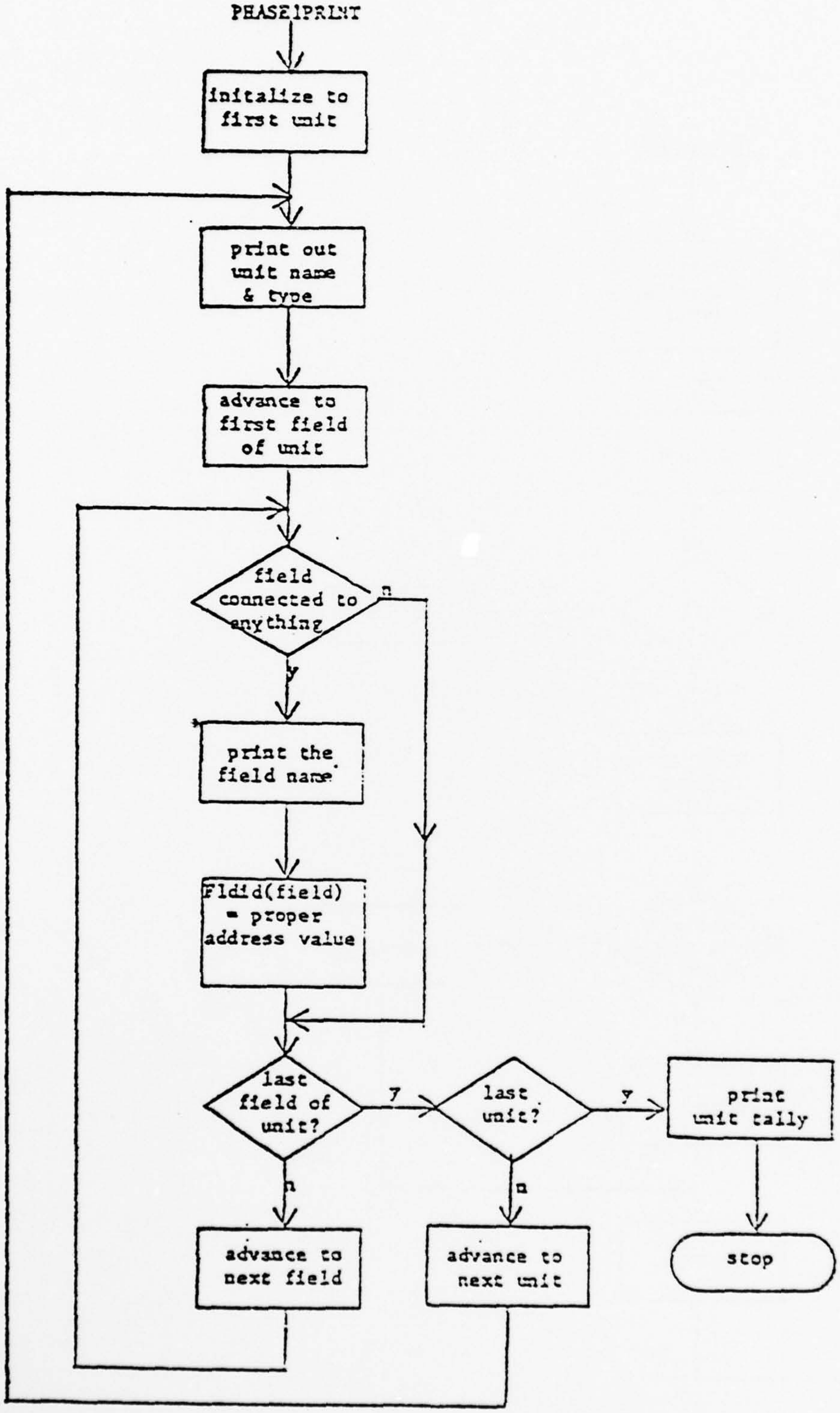

A
initialize variables OPENFILE (phase2file)
GETLINE
eof?
y
return
n
GETNAME
label?
y
GETNAME
n
memory reference or branch inst
n
y
process condition code & indirection if present
process label

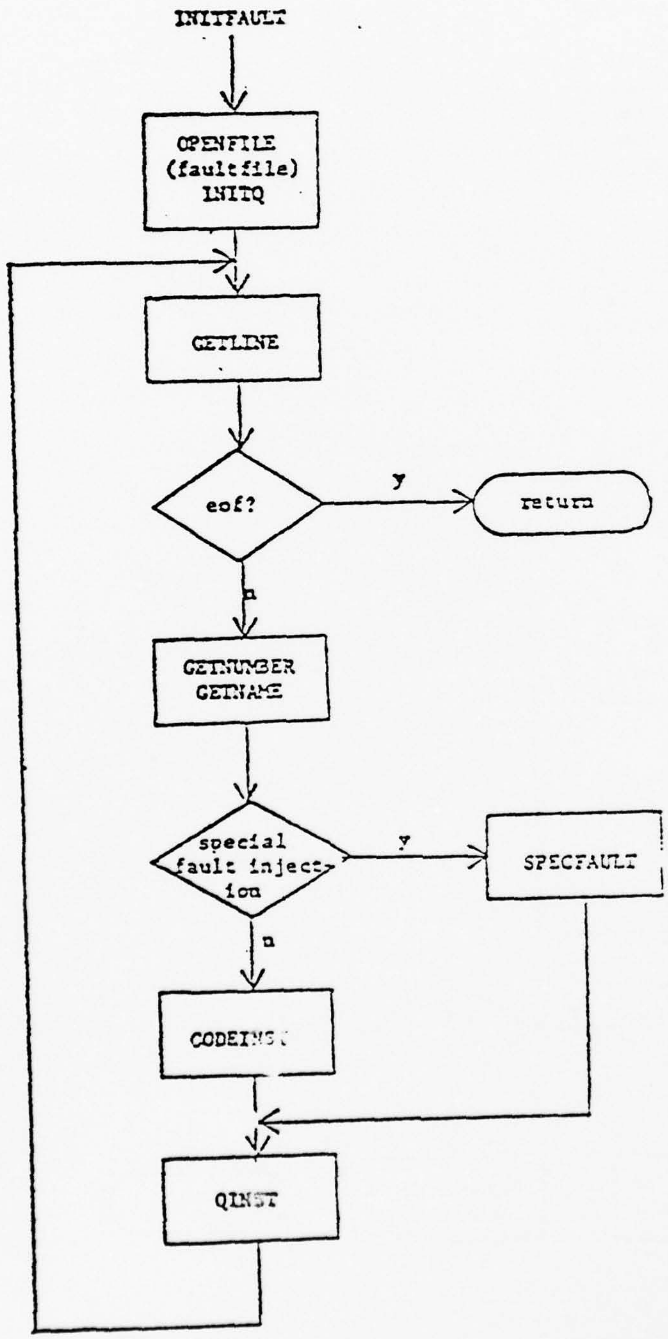

MAIN3
clock = 0
pa = beg. of program
pa < progend
n
return
y
KEVENTS(clock)
PARSEWORD(pa)
QINST
Increment pa
faultbit in instr. set
n
y
clock =+ decodetime

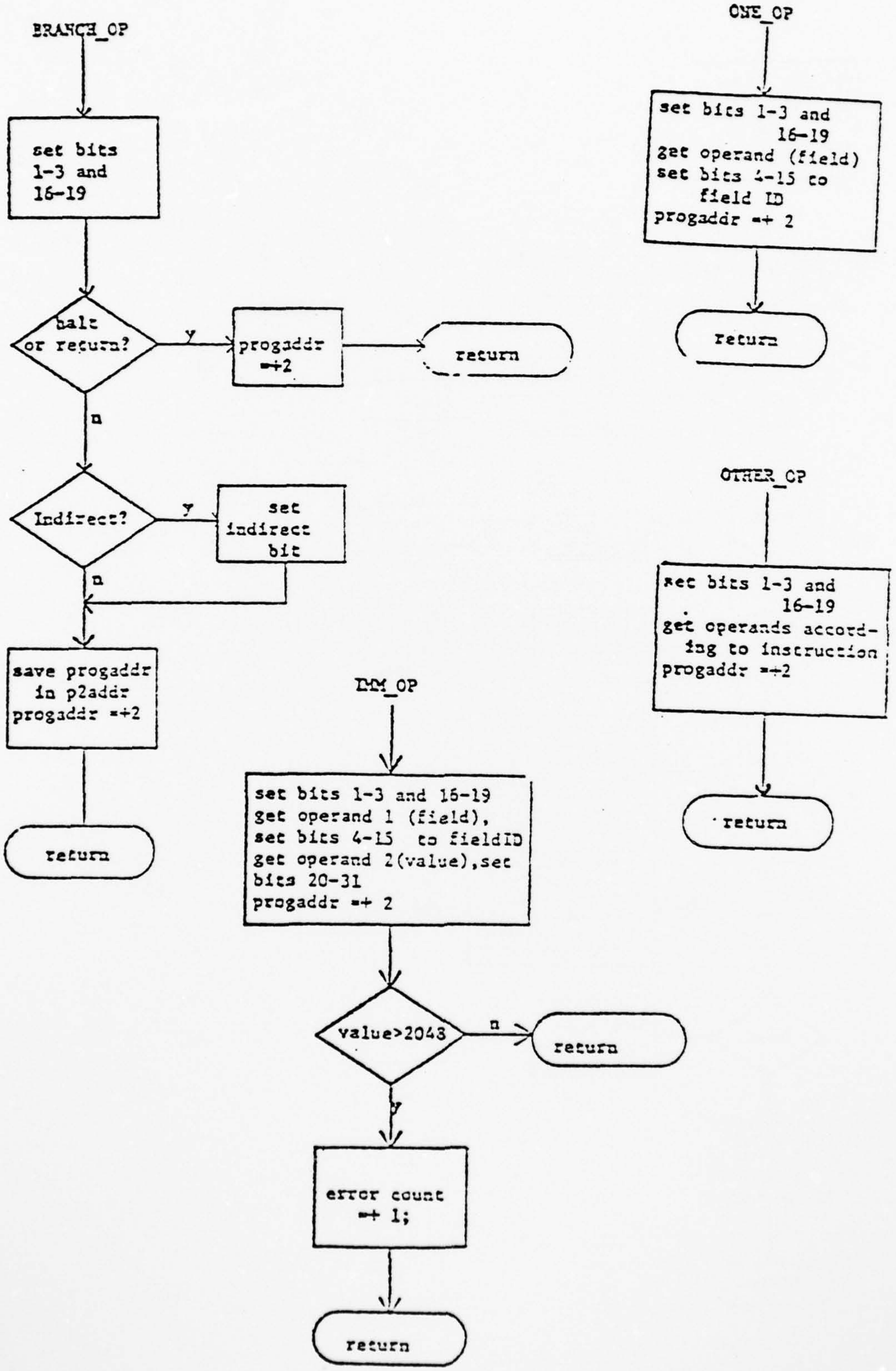

FINDNAME
scan all the defined existing units for the name passed to this routine
name found?
n
return(0)
y
return (unittype)

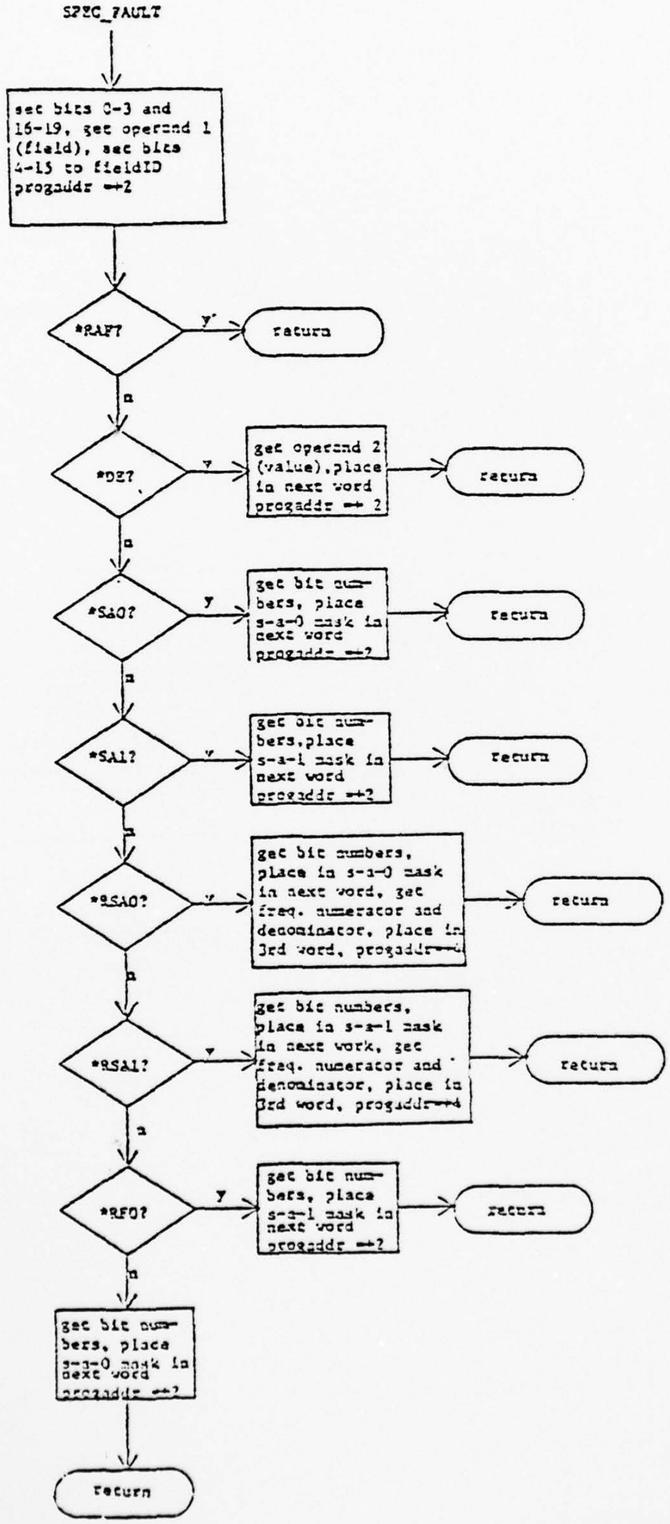

CODEINST
Call appropriate routine according to instruction number:
TWO_OP
IMM_OP
BRANCH_OP
ONE_OP
UNIT_OP
OTHER_OP
return

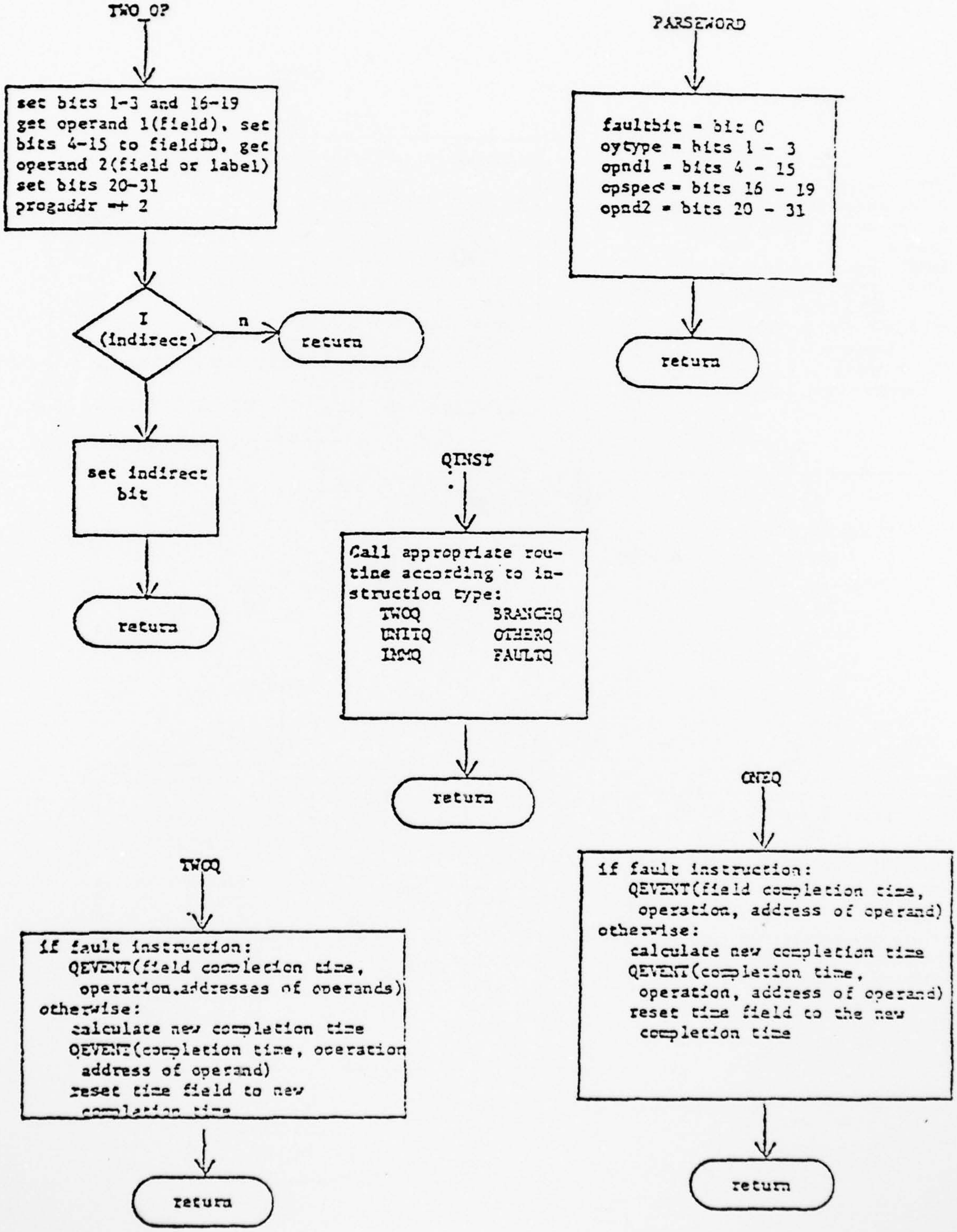

ENTERUNIT
enter the name of the specified unit type in the data structures
enough room?
n
return(0)
y
return(1)

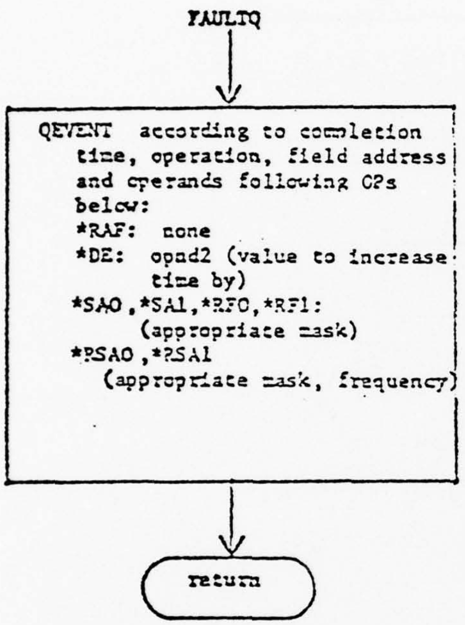

PHASE1PRINT
initalize to first unit
print out unit name & type
advance to first field of unit
field connected to anything
n
y
print the field name
Fldid(field) = proper address value
last field of unit?
y
n
last unit?
y
n
print unit tally
stop
advance to next field
advance to next unit


INITFAULT
OPENFILE
(faultfile)
INITQ
GETLINE
eof?
y
return
n
GETNUMBER
GETNAME
special
fault injection
y
SPECFAULT
n
CODEINST
QINST

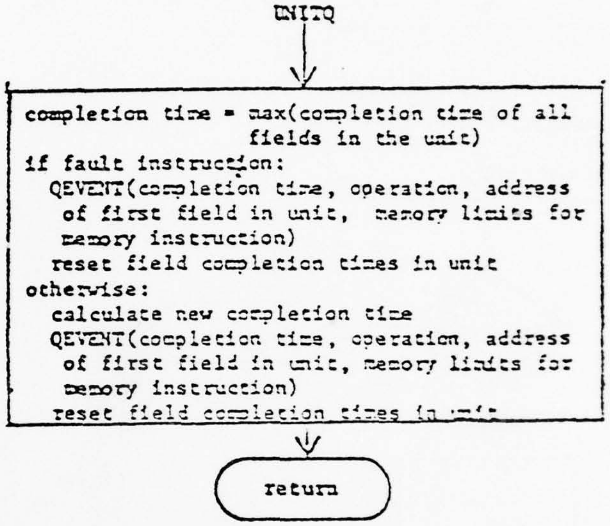

BRANCH_OP
set bits 1-3 and 16-19
halt or return?
y
progaddr =+2
return
n
Indirect?
y
set indirect bit
n
save progaddr in p2addr progaddr =+2
return
ONE_OP
set bits 1-3 and 16-19 get operand (field) set bits 4-15 to field ID progaddr =+ 2
return
OTHER_OP
set bits 1-3 and 16-19 get operands according to instruction progaddr =+2
return
IMM_OP
set bits 1-3 and 16-19 get operand 1 (field), set bits 4-15 to fieldID get operand 2(value),set bits 20-31 progaddr =+ 2
value>2043
n
return
y
error count =+ 1;
return


SPEC_FAULT
set bits 0-3 and 16-19, get operand 1 (field), set bits 4-15 to fieldID progaddr →+2
*RAF?
y
return
n
*DE?
y
get operand 2 (value), place in next word progaddr → 2
return
n
*SA0?
y
get bit numbers, place s-a-0 mask in next word progaddr →+2
return
n
*SA1?
y
get bit numbers, place s-a-1 mask in next word progaddr →+2
return
n
*RSA0?
y
get bit numbers, place in s-a-0 mask in next word, get freq. numerator and denominator, place in 3rd word, progaddr→+4
return
n
*RSA1?
y
get bit numbers, place in s-a-1 mask in next work, get freq. numerator and denominator, place in 3rd word, progaddr→+4
return
n
*RFO?
y
get bit numbers, place s-a-1 mask in next word progaddr →+2
return
n
get bit numbers, place s-a-0 mask in next word progaddr →+2
return

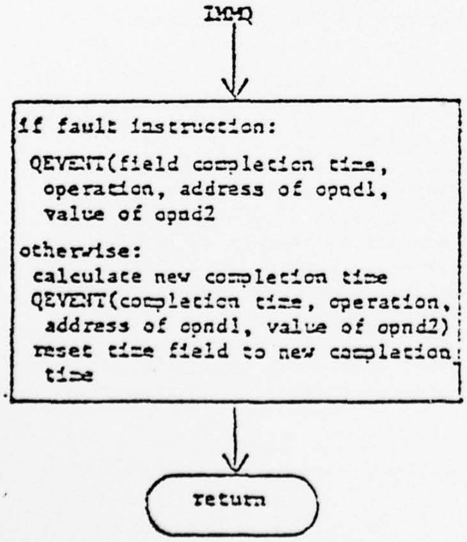

TWO_OP
set bits 1-3 and 16-19
get operand 1(field), set
bits 4-15 to fieldID, get
operand 2(field or label)
set bits 20-31
progaddr =+ 2
I
(indirect)
n
return
set indirect
bit
return
PARSEWORD
faultbit = bit 0
oytype = bits 1 - 3
opnd1 = bits 4 - 15
opspec = bits 16 - 19
opnd2 = bits 20 - 31
return
QINST
Call appropriate rou-
tine according to in-
struction type:
TWOQ
BRANCHQ
UNITQ
OTHERQ
IMMQ
FAULTQ
return
ONEQ
if fault instruction:
QEVENT(field completion time,
operation, address of operand)
otherwise:
calculate new completion time
QEVENT(completion time,
operation, address of operand)
reset time field to the new
completion time
return
TWOQ
if fault instruction:
QEVENT(field completion time,
operation,addresses of operands)
otherwise:
calculate new completion time
QEVENT(completion time, operation
address of operand)
reset time field to new
completion time
return



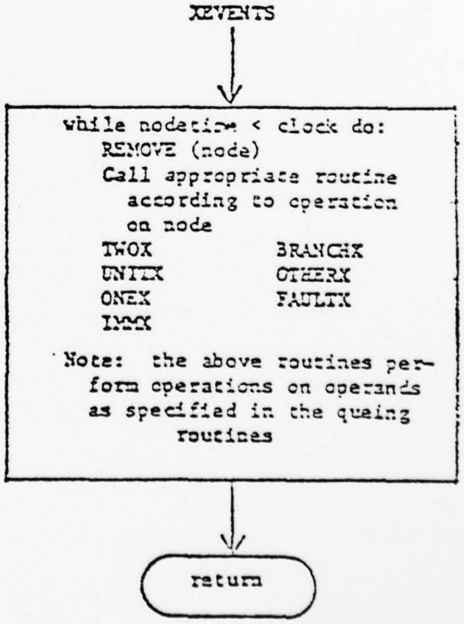
```
QEVENT  according to completion
   time, operation, field address
   and operands following OPs
   below:
   *RAF:  none
   *DE:  opnd2 (value to increase
         time by)
   *SAO,*SA1,*RFO,*RF1:
         (appropriate mask)
   *PSAO,*PSA1
      (appropriate mask, frequency)
```



```
completion time = max(completion time of all
                      fields in the unit)
if fault instruction:
  QEVENT(completion time, operation, address
   of first field in unit,  memory limits for
   memory instruction)
  reset field completion times in unit
otherwise:
  calculate new completion time
  QEVENT(completion time, operation, address
   of first field in unit, memory limits for
   memory instruction)
  reset field completion times in unit
```



```
if fault instruction:
 QEVENT(field completion time,
  operation, address of opnd1,
  value of opnd2
otherwise:
 calculate new completion time
 QEVENT(completion time, operation,
  address of opnd1, value of opnd2)
 reset time field to new completion
  time
```



```
while nodetime < clock do:
   REMOVE (node)
   Call appropriate routine
     according to operation
     on node
   TWOX            BRANCHX
   UNITX           OTHERX
   ONEX            FAULTX
   IMMX
Note:  the above routines per-
  form operations on operands
  as specified in the queing
        routines
```

4. DESCRIPTION OF LANGUAGE

This chapter presents a syntactic description of the languages used in Phase-I and Phase-II of the simulator. The language will be presented using a format similar to the commonly-used Backus Normal Form. Sequences of characters enclosed in "()" represent entities whose values are strings of symbols. An example of a syntactic rule is the following:

```
(ab) :=  c  |  (ab)
```

where the mark ":=" means "is defined as" and the mark "|" means "OR". Entities enclosed in "[ ]" indicate that the entities are optional.

The following syntactic rules apply to programs in both Phase-I and Phase-II.

```
(digit) := 0  |  1  |  2  |  3  |  4  |  5  |
           6  |  7  |  8  |  9

(letter) := a  |  b  |  c  |  d  |  e  |  f  |
            g  |  h  |  i  |  j  |  k  |  l  |
            m  |  n  |  o  |  p  |  q  |  r  |
            s  |  t  |  u  |  v  |  w  |  x  |
            y  |  z  |  A  |  B  |  C  |  D  |
            E  |  F  |  G  |  H  |  I  |  J  |
            K  |  L  |  M  |  N  |  O  |  P  |
            Q  |  R  |  S  |  T  |  U  |  V  |
            W  |  X  |  Y  |  Z

(identifier) := (letter)  |  (identifier)(letter)  |
                     (identifier)(digit)
(number) := [-](digit)  |  (number)(digit)
(numbers) := (number)  |  (number)(,numbers)
```

Any mark in a formula which is not a variable or a connective denotes itself.

4.1 Phase-I: Definition Phase

Syntactic Description:

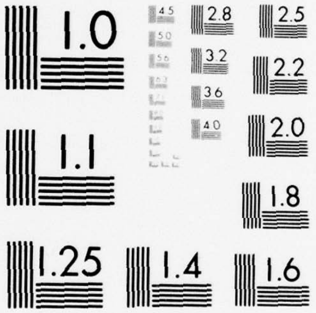
```
(ALU field) := r1  |  r2  |  m  |  d  |  s

(bus field) := s  |  d  |  a

(CPU field) := m  |  s  |  r1  |  r2  |  r3  |
               r4  |  r5  |  r6  |  r7  |  r8

(memory field) := a  |  s  |  d

(VSD field) := m  | d1  | d2  |  d3  |  d5  |
               d6  |  d7  |  d8  |  s  |  d

(peripheral device) :=  m  |  d1  |  s  |  d

(field identifier := (identifier).a(ALU field)  |
                    (identifier).b(bus field)  |
                    (identifier).c(CPU field)  |
                    (identifier).m(memory field)  |
                    (identifier).v(VSD field)  |
                    (identifier).p(peripheral field)

(field identifier list) := (field identifier)  |
                           (,field identifier list)

(command line)  := (field identifier)>(field
                  identifier list)

(program) := (command line)  |
          (program)(command line)
```

Additional rule: a (field identifier) may appear only once on the left of the ">" of the program. Future references of that (field identifier) on the left of the ">" will be ignored.

AD-A063 799 RESEARCH TRIANGLE INST RESEARCH TRIANGLE PARK NC SYST--ETC F/G 9/2
BASIC RESEARCH IN SUPPORT OF CONCURRENT FAULT MONITORING IN MOD--ETC(U)
JUN 78 J W GAULT, P N MARINOS, K S TRIVEDI N00039-77-C-0363
UNCLASSIFIED RTI/1504/00-01-1 NL

4 OF 4
AD
A063799

END
DATE
FILMED
3-79
DDC


1.0
4.5
5.0
5.6
6.3
7.1
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

4.2 Phase-II: Assembler Phase

4.2.1 Syntactic Description of Language

A program is a sequence of statements. Statements are separated one from the next by the end of line character, i.e., each statement is placed on a single, separate line.

The definitions of "identifier" and "field identifier" in this section will be as defined in Section 4.1.

A statement, S, may be labeled as follows:

```
              (identifier):  S
```

Syntactic Description of Assembler File:

```
    (unit identifier) :=  (identifier)
    (label) := (identifier)
    (statement) := (op1) (field identifier 1),
                          (field identifier 2)[,I]  |
                   (op2) (field identifier 1),(label)  |
                   (op3) (unit identifier)  |
                   (op4) (unit identifier),(number),(number)  |
                   (op5) (field identifier)  |
                   (op6) (field identifier),(number)  |
                   (op7) (label)[,I]  |
                   (op 8)  (number),(label)[,I]
```

```
                    (op9)   |

                    (op10)  (unit identifier),([I][O])   |

                    (op11)  (field identifier)   |

                    (op12)  (field identifier),(number)   |

                    (op13)  (field identifier),(numbers)   |

                    (op14)  (field identifier),(numbers);
                                  (number)/(number)

              (note:  operands for (op1) and (op14)
                      should be on the same line)

      (op1)  := AD   |   SB   |   M   |   D   |   AN   |   OR   |
                XR   |   MV   |   C   |   *AD   |   *SB   |   *M   |
                *D   |   *AN   |   *OR   |   *XR   |   *MV   |   *C

      (op2)  :=  LDM   |   STM   |   *STM   |   *LDM

      (op3)   :=   ADU   |   SBU   |   MPU   |   DU   |   ANU   |
                   ORU   |   XPU   |   CLU   |   UDUMP    |   VOTE   |
                   SST   |   RST
                   *ADU   |   *SBU   |   *MPU   |   *DU   |   *ANU   |
                   *ORU   |   *XRU   |   *CLU   |   *UDUMP   |
                   *VOTE   |   *SST   |   *RST

      (op4)   :=   MDUMP   |   *MDUMP

      (op5)  := NEG   |   NOT   |   SQRT   |   ABS   |
                CL   |   PAR   |   *NEG   |   *NOT   |   *SQRT   |
                *ABS   |   *CL   |   *PAR

      (op6)   :=   ADI   |   SBI   |   MI   |   DI   |   ANI   |
                   ORI   |   XRI   |   LI   |   CI   |   SLL   |
                   SRL   |   SLA   |   SRA   |
                   *ADI   |   *SBI   |   *MI   |   *DI   |   *ANI   |
                   *ORI   |   *XRI   |   *LI   |   *CI   |   *SLL   |
                   *SRL   |   *SLA   |   *SRA

      (op7)   :=   B      |   BSA   |   *B    |   *BSA

      (op8)   :=   BC   |   *BC

      (op9)   :=   HLT   |   RET   |   *HLT   |   *RET   |
                   BOMB   |   *BOMB

     (op10)   :=   IO   |   *IO
```

```
(op11)  :=  *RAF

(op12)  :=  *DE

(op13)  :=  *SA0, *SA1, *RF0, *RF1

(op14)  :=  *RSA0, *RSA1
```

Comments:

Comments may follow the operands of instructions provided the operands are followed by one or more blanks and a semicolon. Random-stuck-at instructions are the only exceptions and may not be followed by comments.

Pseudos:

Pseudo-operations are of the following form:

```
$DEF    number(,numbers)
           where numbers are as specified above

$DEFO   number(,numbers)
           where numbers are as specified above
           except that the digits must be integers
           from 0 (zero) to 7 (seven).  The
           octal value is used here.
```

Syntactic Description of Fault Initialization File:

A statement is as defined above with the addition of a mandatory number (the time a statement is to be executed) preceding the instruction mnemonic. Also, the first character of each instruction mnemonic must begin with "*", i.e., must be a fault-injection instruction. We have the following definition:

(timing statement) := (number) (statement)

where (statement) is defined as indicated in the assembler file (with mnemonics required to begin with "*"), and the (number) specifying the time the instruction is to be executed. The (number) may not be preceded by a minus sign (-) in this statement, i.e., all numbers must be non-negative.

4.2.2 Language Mnemonics and their Description

In the following discussion, the notation "<-" indicates "becomes", for example, "NUM1 <- NUM1 + NUM2" states that the value of NUM1 is changed to the sum of NUM1 and NUM2. The notation "c(NAME)" refers to the contents of the field or address with representation NAME, the notation "?" indicates a comparison operation, the notation "<<" indicates a left shift, and the notation ">>" indicates a right shift. "OP" represents the instruction mnemonics for a particular instruction type.

Two operand instructions:

Type I: OP (field identifier 1),(field name 2)

Let v1 = c(field identifier 1)
v2 = c(field identifier 2), if no I is present
c(c(field identifier 2)), if I is present

OP(mnemonic)	Operation
[*]AD	v1 <- v1 + v2
[*]SB	v1 <- v1 - v2
[*]M	v1 <- v1 * v2
[*]D	v1 <- v1 / v2
[*]AN	v1 <- v1 & v2
[*]OR	v1 <- v1 | v2 (inclusive or)
[*]XR	v1 <- v1 ¬ v2 (exclusive or)
[*]MV	v1 <- v2
[*]C	v1 ? v2, set condition code register

Type II: OP (field identifier),(label)[,I]

Let v1 = c(field identifier)
v2 = c(label) if no I is present
c(c(label)) if I is present

OP(mnemonic)	Operation
[*]LDM	v2 <- v1
[*]STM	v1 <- v2

Unit Operand Instuctions:

Type I: OP (unit identifier)

Note: the unit field acronyms are used in the table below to indicate the operands actually used by the operations (see Section 3.2.1).

OP(mnemonic)	Unit type	Operation
[*]ADU	ALU	d <- r1 + r2
[*]SBU	ALU	d <- r1 - r2
[*]MPU	ALU	d <- r1 * r2
[*]DU	ALU	d <- r1 / r2
[*]ANU	ALU	d <- r1 & r2
[*]ORU	ALU	d <- r1 | r2 (inclusive or)
[*]XRU	ALU	d <- r1 ¬ r2 (exclusive or)
[*]CLU	any	clears all fields of unit
[*]UDUMP	any	prints all fields of unit
[*]VOTE	VSD	d <- word representing the majority of registers
[*]SST	any	s <- 1
[*]RST	any	s <- 0

Type II: [*]MDUMP (unit identifier)(number1),(number2)

-prints memory words from location (number 1) to location (number 2)
Note: (number 1) < (number 2)

Single operand instructions:

OP (field identifier)

Let v1 = c(field identifier)

OP(mnemonic)	Operation
[*]NEG	v1 <- twos complement of v1
[*]NOT	v1 <- ones complement of v1
[*]SQRT	v1 <- square root of v1
[*]ABS	v1 <- absolute value of v1
[*]CL	v1 <- 0
[*]PAR	parity of v1 checked, condition code register set

Immediate Operand Instructions:

```
OP  (field identifier),(number)

Let v1 = c(field identifier)
    v2 = (number)
```

OP(mnemonic)	Operation
[*]ADI	v1 <- v1 + v2
[*]SBI	v1 <- v1 - v2
[*]MI	v1 <- v1 * v2
[*]DI	v1 <- v1 / v2
[*]ANI	v1 <- v1 & v2
[*]ORI	v1 <- v1 | v2 (inclusive or)
[*]XRI	v1 <- v1 ¬ v2 (exclusive or)
[*]LI	v1 <- v2
[*]CI	v1 ? v2, set condition code register
[*]SLL	v1 <- v1 << v2 (logical)
[*]SRL	v1 <- v1 >> v2 (logical)
[*]SLA	v1 <- v1 << v2 (arithmetic)
[*]SRA	v1 <- v1 >> v2 (arithmetic)

Other Instructions:

```
[*]IO    (unit identifier),([I,O])

   Operation:  If I:  Input into unit
               If O:  Output from unit

[*]BOMB

   Operation:  program immediately stops,
               instructions remaining on the
               event queue are not executed.
```

Branch instructions:

Type I: OP (label)[,I]

Let a1 = address of label if I not present,
c(address of label) if I present

OP(mnemonic)	Operation
[*]B	program counter <- a1
[*]BSA	program counter placed on stack, program counter <- a1

Type II: BC (number),(label)[,I]

Let a1 be defined as above.

number = a condition code number
The condition codes are indicated as follows
(leftmost bit is bit 0):

Condition	Bit number set	Condition Code Number
equal	31	01
less than	30	02
greater than	29	04
parity (odd)	28	08
parity (even)	27	016

Operation: the condition code number is compared to the condition code register. If any of the logical 1 bits of the values compared match, the branch is performed as in Type I. Otherwise, no branch is performed.

Type III: OP

OP(mnemonic)	Operation
[*]HLT	branch to end of program
[*]RET	pop address from stack, pc <- address + 1 (next word)

Special Fault Injection Instructions:

Type I: *RAF (field identifier)

Operation: error register associated with
(field identifier) is set to 0
s-a-1 masks set to logical 0's
s-a-0 masks set to logical 1's

Type II: *DE (field identifier),(number)

Operation: adds (number) to the last completion time register associated with (field identifier)

Type III: OP (field identifier),(numbers)

OP (mnemonic) Operation

*SA0

bits corresponding to (numbers) are assigned logical 0 in s-a-0 mask associated with (field identifier)
error register "bit 31" set to 1

*SA1

bits corresponding to (numbers) are assigned logical 0 in s-a-1 mask associated with (field identifier)
error register "bit 30" set to 1

*RF0

bits corresponding to (numbers) are assigned logical 1 in s-a-0 mask associated with (field identifier)
if s-a-0 masks bits are all logical 1, appropriate bits in error register are set to 0

*RF1

bits corresponding to (numbers) are assigned logical 0 in s-a-1 mask associated with (field identifier)
if s-a-1 masks bits are all logical 0, appropriate bits in error register are set to 0

Special Fault Injection Instructions (cont):

Type IV: OP (field identifier),(numbers);
(number1)/(number2)

(note: instruction should be on a single line)

OP(mnemonic)	Operation
*RSA0	same operations as performed in *SA0 except "bit 29" is set to 1; additionally, frequency = (number1) divided by (number2) is placed in frequency field associated with field identifier
*RSA1	same operations as performed in *SA1 except "bit 28" is set to 1; additionally, frequency = (number1) divided by (number2) is placed in frequency field associated with field time

5. EXAMPLE

One possible use of the simulator is illustrated in the following example where the fault-tolerance of a system is examined subject to the injection of faults at different arrival rates. Interarrival times of the faults were exponentially distributed with the following three arrival rates:

rate 1: 1/2000

rate 2: 1/5000

rate 3: 1/10000

The system studied is illustrated in Figure 5.1, with the Phase-I input file which created the system listed in Figure 5.2. To inject the desired faults, a program was written which generated the fault-initialization file to inject the faults according to exponential arrival times; an example is shown in Figure 5.3. The Phase-II program which is listed in Figure 5.4 loops twenty times, performing a simple arithmetic calculation in each of the six arithmetic-logic-units, and voting twice for verification. With no faults injected, this program required 17,067 clock units to complete using operation times listed in Figure 5.5 (from the DEFINE file).

To study the fault tolerance of the system, one hundred runs were made with each arrival rate and the number of "failures" and "successes" recorded. A failure was defined as "no majority value in the vsd", or "improper time to complete the Phase-II program", i.e., a time other than 17,067 time units. A successful run was defined as one which did not fail.

The results of the computer runs are listed in Figure 5.6. As would be expected, the decreasing arrival rate of faults resulted in a greater number of successful completions of the program. The tolerance of the system according to these arrival rates is thus illustrated.

It should be noted that this example serves to illustrate only one use of the simulator. Many other uses as mentioned earlier are also supported.

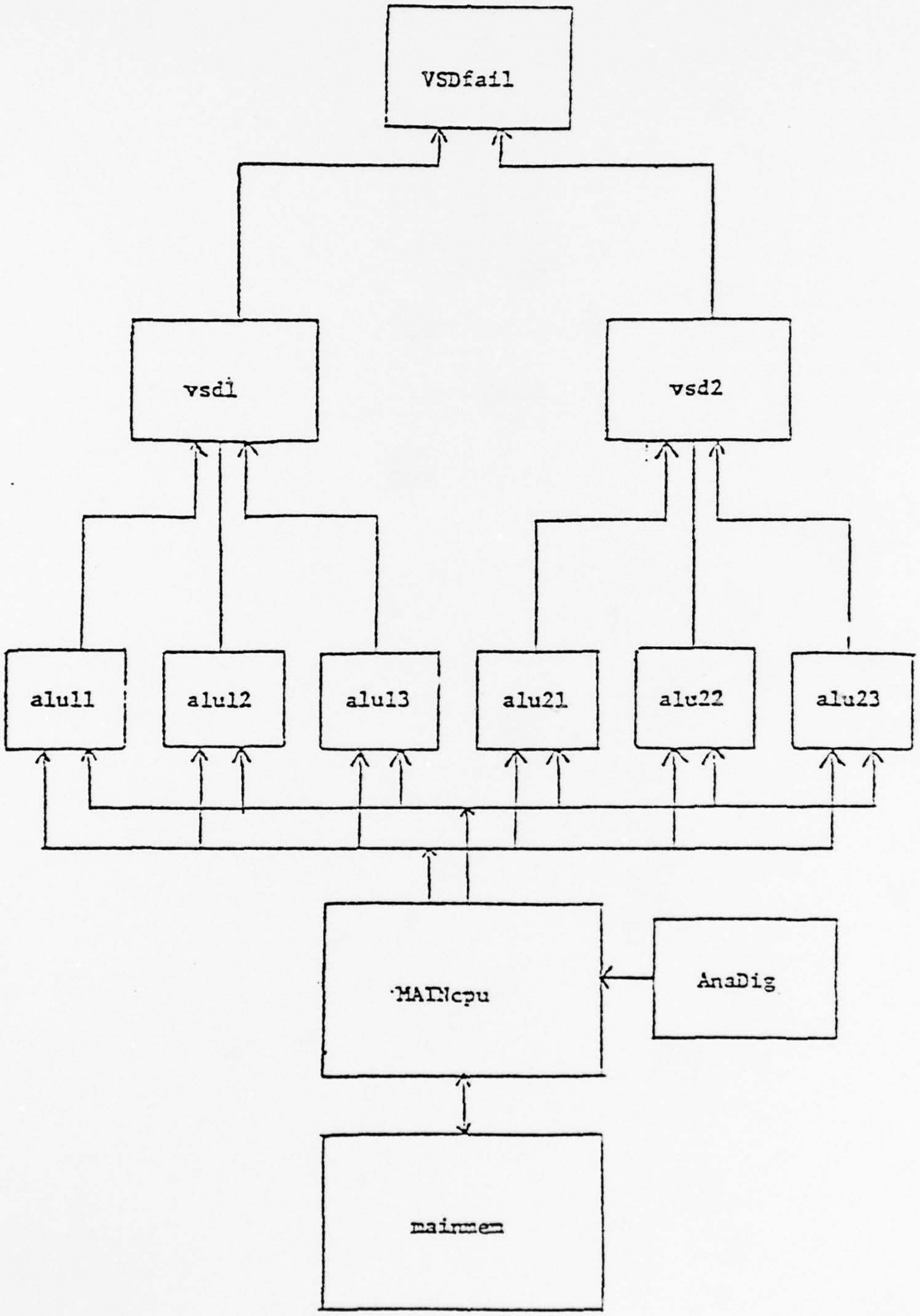


Figure 5.1: System configuration created by the Phase-I input file listed in Figure 5.2.

```
MAINcpu.cr1  > mainmem.md
MAINcpu.cr8     >
                        alu21.ar1,
                        alu22.ar1,
                        alu23.ar1,
                        alu11.ar1,
                        alu12.ar1,
                        alu13.ar1
MAINcpu.cr7     >
                        alu21.ar2,
                        alu22.ar2,
                        alu23.ar2,
                        alu11.ar2,
                        alu12.ar2,
                        alu13.ar2
alu21.ad > vsd2.vd1
alu22.ad>vsd2. vd2
alu23.ad>       vsd2.vd3
alu11.ad>vsd1. vd1
alu12.ad>vsd1. vd2
alu13.ad>vsd1. vd3
vsd1.vd >  VSDfail.vd1
vsd2.vd >       VSDfail. vd2
AnaDig.pd1 > MAINcpu.cr2
```

Figure 5.2: Phase-I input used to create the system in Figure 5.1.

```
926       *RSA1    AnaDig.pd,31,24,5;  1/74
5663      *NCT    MAINcpu.cr8
7043      *SA1    alu12.ad,26,15,8
13263     *XRI    MAINcpu.cr2,3538
20592     *SEU    alu13
27122     *RSA0    alu22.ar2,5;  1/98
28016     *SA1    mainmem.md,29,14,19
29049     *DE    MAINcpu.cr1,21141
33173     *SA0    alu12.ar1,0
34488     *SA0    AnaDig.pd,31,24,5,22
34908     *SA0    MAINcpu.cr8,29,14
36123     *SA1    MAINcpu.cr8,21
40458     *SA0    alu12.ar1,0
42291     *RSA0    alu22.ar1,31;  1/68
42953     *RSA0    mainmem.md,23,16,29,14;  1/67
44498     *DU    alu12
46537     *SA0    alu23.ar2,18,7,0,13
48222     *CL    mainmem.ma
53450     *RSA1    AnaDig.pd,1,2,23;  1/36
55310     *LI    MAINcpu.cr2,442
56318     *SRA    alu13.ar1,20
57057     *SA0    MAINcpu.cr6,3,12,9,10
58285     *RSA1    mainmem.md,1,2,23;  1/50
58636     *SA1    alu22.ar2,15
59994     *LI    AnaDig.pd1,3921
```

Figure 5.3: Sample fault-initialization program with exponential interarrival times.

```
        LI      vsd1.vm, 3
        LI      vsd2.vm, 3
        LI      VSDfail.vm, 2
        LI      AnaDig.pm, 1
        LI      MAINcpu.cr4, 20
loop:   IO      I, AnaDig
        MV      MAINcpu.cr2, AnaDig.pd1
        IO      I, AnaDig
        MV      MAINcpu.cr3,AnaDig.pd1
        MI      MAINcpu.cr2, 102
        MI      MAINcpu.cr3, 500
        MV      alu11.ar1, MAINcpu.cr2
        MV      alu12.ar1, MAINcpu.cr2
        MV      alu13.ar1, MAINcpu.cr2
        MV      alu21.ar1, MAINcpu.cr2
        MV      alu22.ar1, MAINcpu.cr2
        MV      alu23.ar1, MAINcpu.cr2
        MV      alu11.ar2, MAINcpu.cr3
        MV      alu12.ar2, MAINcpu.cr3
        MV      alu13.ar2, MAINcpu.cr3
        MV      alu21.ar2, MAINcpu.cr3
        MV      alu22.ar2, MAINcpu.cr3
        MV      alu23.ar2, MAINcpu.cr3
        ORU     alu11
        ORU     alu12
        ORU     alu13
        ORU     alu21
        ORU     alu22
        ORU     alu23
        MV      vsd1.vd1, alu11.ad
        MV      vsd1.vd2, alu12.ad
        MV      vsd1.vd3, alu13.ad
        MV      vsd2.vd1, alu21.ad
        MV      vsd2.vd2, alu22.ad
        MV      vsd2.vd3, alu23.ad
        VOTE    vsd1
        VOTE    vsd2
        MV      VSDfail.vd1, vsd1.vd
        MV      VSDfail.vd2, vsd2.vd
        VOTE    VSDfail
        CI      VSDfail.vd, 502
        BC      6, bomb
        SBI     MAINcpu.cr4, 1
        CI      MAINcpu.cr4, 0
        BC      6, loop
        HLT
bomb:   BOMB
```

Figure 5.4: Phase-II input used to program the system.

```
define                  ADDTIME           10
define                  MULTIME           15
define                  DIVTIME           20
define                  ANDTIME           3
define                  ORTIME            3
define                  XORTIME           5
define                  MOVETIME          3
define                  CPRTIME           25
define                  DECODETIME        1
define                  NEGTIME           3
define                  CPLTIME           3
define                  SQRTIME           45
define                  ABSTIME           12
define                  CLRTIME           6
define                  PARTIME           6
define                  LOADTIME          3
define                  LSHTIME           15
define                  ASHTIME           20
define                  BCHTIME           3
define                  IOTIME            350
define                  DUMPTIME          20
define                  VOTETIME          22
define                  FETCHTIME         20
```

Figure 5.5: Operation times as specified in the define file of the example.

arrival rate of faults	number of successful completions of 100 possibilities
1/2000	75
1/5000	89
1/10000	93

Figure 5.6: Number of successful completions of the program listed in Section 5.4 subject to exponential interarrival times with arrival rates as specified.

6. CONCLUSION

The simulator described in this thesis provides a means of studying the reliability of many system configurations subject to any number of fault distributions. It provides a means of defining a system by using a set of unit types, and a means of studying the reliability of the systems defined by supporting the injection of faults and providing a means of programming error-detection and error-recovery.

The simulator thus combines the benefits of real hardware organizations and analytical models by supporting the modeling of faults in a wide range of systems and facilitating performance evaluation based on the processing of typical workloads subject to these faults.

BIBLIOGRAPHY

1. Avizienis, A. A., et. al. "The STAR Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design": *IEEE Trans. Comput.* C-22, 11 (November 1971), 1312-1321.

2. Backus, J. W., "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. *Proc. Int. Conf. Information Processing* (June 1959), 125-132.

3. Breuer, Melvin A. and Arthur D. Friedman, *Diagnosis and Reliable Design of Digital Systems*. Woodland Hills, Calif: Computer Science Press, Inc, 1976.

4. Hogg, Robert V. and Allen T. Craig, *Introduction to Mathematical Statistics*. London: The Macmillan Company, 1970.

5. Kleinrock, Leonard, *Queueing Systems Volume I: Theory*. New York: John Wiley and Sons, 1975.

6. Knuth, Donald E., *The Art of Computer Programming, Volume 1*. Menlo Park, Calif: Addison-Wesley Publishing Company, 1973.

7. Lavington, S. H. and T. Viswanathan, "A Diagnostic and Reconfiguration System for the MU5 Multicomputer Complex": *Proc. Int. Symp. Fault Tolerant Computing* (1976), 176-181.

8. Levy, H. Olivier and Ralph B. Conn, "A Simulation Program for Reliability Prediction of Fault Tolerant Systems": *Proc. Int. Fault Tolerant Computing* (1975), 104-109.

9. Smith, Charles O., *Introduction to Reliability in Design*. New York, New York: McGraw-Hill Book Company, 1976.

10. Tasar, O., and V. Tasar, "Study of Intermittent Faults in Computers": *AFIPS* 46 (1977), 807-812.

11. Wachter, Wilfred J., "System Malfunction Detection and Correction": *Proc. Int. Symp. Fault Tolerant Computing* (1975), 196-201.

12. Wensley, John H., et. al., "A Comparative Study of Architectures for Fault Tolerance": Proc Int. Symp. Fault Tolerant Computing (1975), 4.16-4.17.

APPENDIX: PROGRAM LISTING

```
include "define"
.struct buf {
        int fildes;
        int nleft;
        char *nextp;
        char buff[512];
        };
struct type0 {
        char    aname[9];

        int     ar1[FLDSIZE],   ar2[FLDSIZE],   am[FLDSIZE],
                ad[FLDSIZE],    as[FLDSIZE];

        };


struct type1 {
        char    mname[9];

        int     ma[FLDSIZE],
                ms[FLDSIZE],
                md[FLDSIZE];

        int     pdpbase;
        };


struct type2 {
        char    cname[9];

        int     cm[FLDSIZE],
                cs[FLDSIZE],
                cr1[FLDSIZE],   cr2[FLDSIZE],   cr3[FLDSIZE],   cr4[FLDSIZE],
                                cr5[FLDSIZE],   cr6[FLDSIZE],   cr7[FLDSIZE],
                                cr8[FLDSIZE];

        };


struct type3 {
        char    bname[9];

        int
                bs[FLDSIZE],
                bd[FLDSIZE],    ba[FLDSIZE];

        };


struct type4 {
        char    vname[9];

        int     vm[FLDSIZE],    vd1[FLDSIZE],   vd2[FLDSIZE],   vd3[FLDSIZE],
                                vd4[FLDSIZE],   vd5[FLDSIZE],   vd6[FLDSIZE],
                                vd7[FLDSIZE],   vd8[FLDSIZE],
                vs[FLDSIZE],    vd[FLDSIZE],    vr1[FLDSIZE];

        };


struct type5 {
        char    pname[9];
        int     pm[FLDSIZE],    pd[FLDSIZE],
                ps[FLDSIZE],    pd1[FLDSIZE];

        };


char *fields[] {
        "ar1",  "ar2",  "am",   "ad",   "as",
        "ma",   "ms",   "md",
        "cm",   "cs",   "cr1",  "cr2",  "cr3",  "cr4",  "cr5",  "cr6",
                "cr7",  "cr8",
        "bs",   "bd",   "ba",
        "vm",   "vd1",  "vd2",  "vd3",  "vd4",  "vd5",  "vd6",  "vd7",
                "vd8",  "vs",   "vd",   "vr1",
        "pm",   "pd",   "ps",   "pd1",
        0
        };


int displace[] {
        0,      1,      2,      3,      4,
        0,      1,      2,
        0,      1,      2,      3,      4,      5,      6,      7,
                8,      9,
        0,      1,      2,
        0,      1,      2,      3,      4,      5,      6,      7,
                8,      9,      10,     11,
        0,      1,      2,      3,
        0
        };


int fldtype[] {
        1,      1,      1,      2,      2,
        1,      2,      3,
        1,      2,      3,      3,      3,      3,      3,      3,
                3,      3,
        2,      3,      3,
        1,      1,      1,      1,      1,      1,      1,      1,
                1,      2,      2,      2,
        1,      1,      2,      2,
        0
        };


char *type[NUMUNITS] {
        "ALU",
        "MEM",
        "CPU",
        "BUS",
        "VSD",
        "PER"
        };


char *errmesg[] {
        "illegal name.field format",
        "unrecognized field name",
        "[ > ] token expected",
        "inconsistant unit-type specification",
        "output field expected before [ > ]",
        "outputs previously defined",
        "input field expected after [ > ]",
        "unit quota filled - see define file for quotas",
        "illegal intra-unit connection",
        "unexpected eof encountered",
        "unidentifiable unit-name",
        "invalid delimiter or improper bit specification format",
        "improper bit specification format--expecting bit number",
        "invalid bit number-- > 31 or < 0 or non-numeric",
        "warning: illegal freq specif--expecting '/', '/' assumed",
        "invalid frequency specification--expecting positive integer",
        "invalid frequency specification--operand missing",
        "invalid delimiter--expecting ',' ",
        "unexpected eof or ';' --missing parameter or operand",
        "invalid parameter--expecting 'I' if any should be present",
        "invalid I/O parameter--expecting 'I' or 'O' ",
        "immediate operand value exceeds range",
        "invalid character or bad name format",
        "unrecognizable instruction",
        "invalid boundary specifications for memory dump",
        "branch to nonexistent label",
        "incorrect or missing unit name",
        "operand for DE instruction must be positive integer",
        "invalid pseudo op",
        "illegal condition code",
        0
        };

struct buf diskbuf, *iobuf;

struct type0 alu[MAXALU], *aluptr, *alutoptr;
struct type1 mem[MAXMEM], *memptr, *memtoptr;
struct type2 cpu[MAXCPU], *cpuptr, *cputoptr;
struct type3 bus[MAXBUS], *busptr, *bustoptr;
struct type4 vsd[MAXVSD], *vsdptr, *vsdtoptr;
struct type5 per[MAXPER], *perptr, *pertoptr;


char line[80], *lineptr, phase1file[72], unitnam[9], fldnam[9];
char phase2file[72];
char faultfile[72];
int *arc[100], *arcptr, topindex[ NUMUNITS ], unitindex, fldisp,
        oldfldisp, sequence, fldindex, baseaddr, memory[ 8192 ];
char  *inst[ ] {
        "AD" ,  /*   0  2-op add*/
        "SB" ,  /*   1   "   subtract  */
        "M"  ,  /*   2   "   multiply  */
        "D"  ,  /*   3   "   divide    */
        "AN" ,  /*   4   "   and       */
        "OR" ,  /*   5   "   or        */
        "XR" ,  /*   6   "   exclusive-or*/
        "MV" ,  /*   7   "   move      */
        "C"  ,  /*   8   "   compare             */
        "LDM",  /*   9   "     load register from memory */
        "STM",  /*  10   "     store from register to memory */
        "ADI",  /*  11  immediate-value add     */
        "SBI",  /*  12          "               */
        "MI" ,  /*  13          "       multiply */
        "DI" ,  /*  14          "       divide   */
        "ANI",  /*  15          "       and      */
        "ORI",  /*  16          "       or       */
        "XRI",  /*  17          "       exclusive or */
        "LI" ,  /*  18          "       load     */
        "CI" ,  /*  19          "       compare  */
        "SLL",  /*  20  shift left logical immediate value */
        "SRL",  /*  21  shift right logical immediate value */
        "SLA",  /*  22  shift left arithmetic immediate value*/
        "SRA",  /*  23  shift right arithmetic immediate value */
        "B"  ,  /*  24  branch unconditionally      */
        "BC" ,  /*  25  branch conditionally */
        "BSA",  /*  26  branch and save return address */
        "HLT",  /*  27  halt */
        "RET",  /*  28  return  */
        "NEG",  /*  29  2's complement */
        "NOT",  /*  30  1's complement */
        "SQRT", /*  31  square root  */
        "ABS",  /*  32  absolute value */
        "CL" ,  /*  33  clear   */
        "PAR",  /*  34  check parity and set parity bit */
        "ADU",  /*  35  unit add  (alu)  */
        "SBU",  /*  36  unit subtract (alu)  */
        "MPU",  /*  37  unit multiply (alu)  */
        "DU" ,  /*  38  unit divide  (alu)   */
        "ANU",  /*  39  unit and     (alu)   */
        "ORU",  /*  40  unit or      (alu)   */
        "XRU",  /*  41  unit exclusive or (alu) */
        "CLU",  /*  42  clear all fields in unit */
        "UDUMP",/*  43  unit dump  */
        "MDUMP",/*  44  memory dump */
        "VOTE", /*  45  vsd activation */
        "SST",  /*  46  set unit status */
        "RST",  /*  47  reset nnit statue */
        "IO",   /*  48  input/output  */
        "BOMB", /*  49  bomb--stop program */
        0
}:
        /*                                                      */
        char *sfinst [] { /* special fault instructions  */
        "*RAF", /*  1  remove all fault injections from field*/
        "*DE" , /*  2  dead end--add specified clock units to
                       event in event queue for field          */
        "*SA0", /*  3  designated bits to be stuck at 0  */
        "*SA1", /*  4  designated bits to be stuck at 1  */
        "*RSA0", /* 5  designated bits to be stuck at 0 at specified
                       interval  */
        "*RSA1", /* 6  designated bits to be stuck at 1 at specified
                       interval  */
        "*RF0", /*  7  remove stuck at 0 faults for designated bits */
        "*RF1", /*  8  remove stuck at 1 faults for designated bits */
        0
}:

char *pseudos[]{
        "DEF",
        "DEFC",
        0
};

char *p2inst[]{ /* instructions requiring processing by pass2 */
        "B",
        "*B",
        "BC",
        "*BC",
        "BSA",
        "*BSA",
        "LDM",
        "*LDM",
        "STM",
        "*STM",
        0
}:
/*  ARRAY REGCODE:  code numbers for bits 16-18 or 16-19 for
    regular instructions and corresponding fault injection
    (arrays INST and FINST above                              */
int regcode[42] {
        000     ,       /* 0 */
        010000  ,       /* 1 */
        020000  ,       /* 2 */
        030000  ,       /* 3 */
        040000  ,       /* 4 */
        050000  ,       /* 5 */
        060000  ,       /* 6 */
        070000  ,       /* 7 */
        0100000,        /* 8 */
        0110000,        /* 9 */
        0120000,        /* 10 */
        000     ,       /* 11 */
        010000  ,       /* 12 */
        020000  ,       /* 13 */
        030000  ,       /* 14 */
        040000  ,       /* 15 */
        050000  ,       /* 16 */
        060000  ,       /* 17 */
        070000  ,       /* 18 */
        0100000,        /* 19 */
        0110000,        /* 20 */
        0120000,        /* 21 */
        0130000,        /* 22 */
        0140000,        /* 23 */
        000     ,       /* 24 */
        020000  ,       /* 25 */
        040000  ,       /* 26 */
        060000  ,       /* 27 */
        0100000,        /* 28 */
        000     ,       /* 29 */
        010000  ,       /* 30 */
        020000  ,       /* 31 */
        030000  ,       /* 32 */
        040000  ,       /* 33 */
        050000  ,       /* 34 */
        000     ,       /* 35 */
        010000  ,       /* 36 */
        020000  ,       /* 37 */
        030000  ,       /* 38 */
        040000  ,       /* 39 */
        050000  ,       /* 40 */
        060000  ,       /* 41 */
        070000  ,       /* 42 */
        0100000,        /* 43 */
        0110000,        /* 44 */
        0120000,        /* 45 */
        0130000,        /* 46 */
        0140000,        /* 47 */
        000     ,       /* 48 */
        020000  ,       /* 49 */
}:

int faultcode[8] {
        000,
        010000,
        020000,
        030000,
        040000,
        050000,
        060000,
        070000
}:

int ones [] { /* to make the ith bit of a word equal to 1,
                OR the word with ones[i]                        */
        0100000,        /* bit 0 */
        040000,         /* bit 1 */
        020000,         /* bit 2 */
        010000,         /* bit 3 */
        04000,          /* bit 4 */
        02000,          /* bit 5 */
        01000,          /* bit 6 */
        0400,           /* bit 7 */
        0200,           /* bit 8 */
        0100,           /* bit 9 */
        040,            /* bit 10 */
        020,            /* bit 11 */
        010,            /* bit 12 */
        04,             /* bit 13 */
        02,             /* bit 14 */
        01,             /* bit 15 */
        0
};

int  zeroes[] {  /*  to zero the ith bit of a word, AND the word
                     with zeroes[i]                                  */
        077777,         /* bit 0 */
        0137777,        /* bit 1 */
        0157777,        /* bit 2 */
        0167777,        /* bit 3 */
        0173777,        /* bit 4 */
        0175777,        /* bit 5 */
        0176777,        /* bit 6 */
        0177377,        /* bit 7 */
        0177577,        /* bit 8 */
        0177677,        /* bit 9 */
        0177737,        /* bit 10 */
        0177757,        /* bit 11 */
        0177767,        /* bit 12 */
        0177773,        /* bit 13 */
        0177775,        /* bit 14 */
        0177776,        /* bit 15 */
        0
};

int     temp,
        errors,  /* keep track of PHASE I errors */
        fldid, /* displacement within structure of given field */
        count, /* temporary counting variable */
        memcount, /* relative address of word of memory being pointed
                        to by progaddr */
        instnum, /* row number in array where instruction is found */
        symaddr[SYMSIZE], /* address of label in corresponding row
                                    of symtab */
        cond, /* where condition code from cond. branch is placed*/
        dummy;
int
        *p2addr[NUMBRANCH], /* pdp addresses of branch instructions'
                                    assembly code */
        **p2adptr, /* pointer to p2addr,points to next unused element*/
        **p2endaddr, /* largest value of **p2adptr  */
        *asymptr, /* pointer to symaddr */
        *memstart, /* PDP address where program binary code begins */
        *progbeg, /* address of beginning of assembled program */
        *progend, /* PDP address where program binary code ends */
        *progaddr;/* pointer to line in memory where code is currently
                being placed or points to next unused word at
                beginning of new instruction    */
int *spare; /*this pointer is going to make everything better--efb*/
char    unitname[9], /*unitname will be placed here*/
```

```
        fieldname[9], /* fieldname will be placed here */
        memload[9], /* name of memory where program will be loaded*/
        symtab[SYMSIZE][9], /* symbol table */
        name[9], /* where a name from input is placed before
                        being deciphered */
        c; /* temporary storage of a character */
char    *mlptr, /* ptr to name of memory read in to memload */
        *symptr, /* pointer to next blank space in symbol table */
        *symend, /* pointer to end of symbol table */
        *nameptr, /* pointer to string name */
        *cptr; /* tempory pointer to character */

int numfields[6]{
        5, /* alu */
        3, /* mem */
        10, /* cpu */
        3, /* bus */
        12, /* vsd */
        4   /* peripheral */
};

/*              PHASE 3 DECLARATIONS                    */


int
        faultbit, /* = 1 if inst being decoded is fault injection,
                        otherwise = 0                          */
        optype,   /* operation type of instruction being decoded */
        opnd1,    /*operand 1 of instruction being decoded */
        opnd2,    /*operand 2 of instruction being decoded */
        opspec,   /*operation specification within type for
                    instruction being decoded                  */
        unittype,
        condcode, /*  condition code register */
        *tptr,  /* temporary pointer */
        initialization, /* if 1, indicates interpreting the
                           fault initialization file, if 0,
                            regular execution */
        dum;
int tvec[2];
long
        longone, /* long one used in assigning value 1 to longs*/
        lcond,
        condtime, /* time field for condit code */
        mtime,
        lz, /* long zero to use in passing to subroutines */
        fintime,  /* completion time of operation being decoded */
        fltime, /*  the time a fault initialization inst in the
                        fault initialization file is to occur*/
        clock;    /* universal clock for all operations*/
extern long seed;
int *regsave[10], /* stack for branch and save */
    rscount;  /*subscript for regsave */
int
        *pa; /* pointer to current line of code being decoded
                        to be placed on event queue */
struct node{
        long            time;
        int             type;
        int             spec;
        char            *add1;
        char            *add2;
        int             val;
        long            mask;
        float           freq;
        struct node     *prev;
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
         struct node      *next;
         int              used;
         };
struct node
         *head, /* pointer to beginning of event queue */
         *tail, /* pointer to end of event queue */
         *p; /* pointer to move down event queue */
struct node
         tailnode,
         headnode;
struct node nodes[NUMNODES]; /* unused nodes to be used for creation
                           of nodes for event queue  */
/*

                      ******MAINLINE******

*/

main()  {
         main1();
         if (errors)  {
                  printf("error detected in PHASE I - aborting\n");
                  goto endprog;
                  }
         else
                  printf("\n\nAll set to proceed with PHASE II\n\n\n");
         main2();
         if(errors)
                  printf("\n*****%d assembler errors, phase 3 skipped",
                           errors);
         else     printf("\n!!!!!!no assembler errors",
                           " phase 3 initiatiated");
         progend = progaddr;
         printf ("\nmemory values are \n");
         progaddr = memstart;
         printf("\n progend = %d, memstart = %d\n",progend,memstart);
         while(progaddr < progend)
         {        printf("   %o   %o\n",progaddr[0],progaddr[1]);
                  progaddr =+ 2;
         }

                  printf("\n\n\n           PHASE THREE \n\n");
         pa = progbeg;
         main3();
endprog:;
}
main1()  {
char     *structptr,      /* general purpose structure pointer */
         outype;          /* output unit type variable */

int      i,j,k,l,m,n,     /* general purpose integer variables */
         comma, *ptr,
         outindex;        /* structure index of the output unit */

printf("\n\n\t\tEE 207 FAULT INJECTION - EVENT DRIVEN SIMULATOR\n\n\n");
printf("PHASE 1 - SYSTEM DESCRIPTION\n\n\n\n");

alutoptr= alu;            /* initialize top pointers */
memtoptr= mem;
cputoptr= cpu;
bustoptr= bus;
vsdtoptr= vsd;
pertoptr= per;

iobuf= &diskbuf;
arcptr= arc;              /* initialize tasks */
```

```
openfile(phase1file);
if (getline() == -1)  goto eof;
for (sequence= 1; j != -1;) {

/* * * * * * * INTERPRETER CODE WHICH OPERATE ACCORDING * * * * * * *
                        TO THE THE ABOVE LOOP
*/

if ((n= getfield( unitnam, fldnam, &fldisp)) == -1)     /* get field*/
                goto eof;
        else if (n == 0) goto erroff;


if ((n= gettok()) == -1) goto eof;                      /* search for '>' */
        else if (n != '>') {
                error(2);
                goto erroff;
                }


if (fldtype[ fldindex ] == 1) {                         /* output fld? */
        error(4);
        goto erroff;
        }


sequence= 1;                                            /* enable error mesgs */


switch (i= findname( unitnam )) {
        case 0:                                         /* new unit */
                if (enterunit (unitnam, fldnam) == 0) {
                        error(7);
                        goto erroff:
                        }
                break;
        default:                                        /* old unit */
                if (fldnam[0] != i) {                   /* consistent? */
                        error (3);
                        goto erroff;
                        }
                if (*(structptr= baseaddr + fldisp + POINTER) != 0) {
                        error (5);                      /* defined before */
                        goto erroff;
                        }
        }


*(structptr= baseaddr + fldisp + POINTER) = arcptr;     /* field address */
outindex= unitindex;
outype= fldnam[0];


for (comma= 0; comma == 0  ||  (j= gettok()) == ',';  comma++) {
        if ((n= getfield( unitnam, fldnam, &fldisp)) == -1) goto eof;
                else if (n == 0) goto erroff;

        if (fldtype[ fldindex ] == 2) {                 /* input fld? */
                error(6);
                goto again;
                }

        switch(k= findname( unitnam )) {                /* defined b4? */
                case 0:                                 /* nope */
                        if (enterunit( unitnam, fldnam) == 0) {
                                error(7);
                        goto again;
                        }
                break;
        default:                                        /* yes */
                if (fldnam[0] != k)  {
                        error(3);
                        goto again;
                        }
        }


        if ((outindex == unitindex) &&
            (outype == fldnam[0]))  {                   /* looping ? */
                error (8);
                goto again;
                }


        *arcptr++= fldnam[0];                           /* pointers into arc */
        *arcptr++= unitindex;
        *arcptr++= baseaddr + fldisp;


        again:;
        }
*arcptr++= 0;
goto loop;

erroff:                 /* here is  the error handling code */
        sequence= 0;            /* disable error messages */
        if (getline() == -1) goto eof;  /* skip the rest of the present line */

loop:;
}
/* * * * * * * * * * HERE IS THE END OF THE INTERPRETER * * * * * * * * * *
*/
eof:


printf("\n\n\n\nCATALOGUE  OF  EXISTING  UNITS\n\n\n");
printf("Name\t Type\tConnected fields\n\n\n");

fldisp= unitindex= 0;           /* initialize */

for (i= 0; i < NUMUNITS; i++)  {        /* unit type loop */
        for (j= 0; j < topindex[ i ]; j++)  {   /* unit index loop */

                switch (i)  {           /* get baseaddr */
                        case 0:
                                baseaddr= &alu[ j ];
                                break;
                        case 1:
                                baseaddr= &aem[ j ];
                                break;
                        case 2:
                                baseaddr= &cpu[ j ];
                                break;
                        case 3:
                                baseaddr= &bus[ j ];
                                break;
                        case 4:
                                baseaddr= &vsd[ j ];
                                break;
                        case 5:
                                baseaddr= &per[ j ];
                                break;
                        }

                printf("%s\t %s\t",baseaddr, type[ i ]);    /* print name & type */
                baseaddr=+ 10;            /* skip name field */
                oldfldisp= fldisp;

                for (fldindex= 0; fldindex == 0  ||
                                displace[ fldisp ] != 0;
                                        fldindex++) {
                        if (connected( fldtype[ fldisp ] ))
                                printf("%s       ",fields[ fldisp ]);
                        *(ptr= baseaddr + FLDID)=
                                (i << 9) + (j << 4) + fldindex;
                        fldisp++;
                        baseaddr= baseaddr + FLDSIZE * 2;
                        }

                fldisp= oldfldisp;
                printf("\n\n");
                unitindex++;
                }

        for (fldisp++; displace [ fldisp ]; fldisp++);
        }

printf("\n\n\nUNIT TALLY\n\n");
for (j= i= 0; i < NUMUNITS; i++) {
        printf("%s\t%d\n\n",type[i],topindex[i]);
        j=+ topindex[i];
        }
printf("\ntotal\t%d\n\n",j);
}
/*
*                       G E T N A M E
*
*       arguments:      target address of name
*       returns:        >0 means all's OK
*                       0  means invalid lead char. in stream
*                       -1 means eof detected
*       notes:          lineptr is left pointing to the first char.
*                       after the last valid name char.
*/

getname (ptr) char *ptr; {
int n,i;
i= ptr;
if (skipspace() != -1) {
while (((n= *lineptr) >= ASC_0  &&  n <= ASC_9) ||
                (n == '*' ) ||
                (n == '$' ) ||
                (n >= ASC_A && n <= ASC_Z)  ||
                (n >= ASC_a && n <= ASC_z)) {
                        lineptr++;
                        if ((ptr- i) < 8) *ptr++ =n;
                        }
        if (ptr == i) return(0); else {
                *ptr= '\0';
                return(1);
                }
        }
return(-1);
}
/*
*                       G E T T O K
*
```

```
 *       arguements:     none.
 *       returns:        the ascii char of an accepted token
 *                       0 if first non-space was not a token
 *                       -1 if an eof was encountered
 *       note:           lineptr is left pointing to the first char
 *                       after an accepted token or at the first
 *                       invalid char.
 */

gettok() {
if (skipspace() != -1)
        switch (*lineptr++) {
                case '>':       return('>');
                case ',':       return(',');
                case '.':       return('.');
                default:        lineptr--;      return(0);
                };
return(-1);
}
/*
 *                       G E T L I N E
 *
 *       arguements:     none.
 *       returns:        zero if all's OK
 *                       -1 on an eof condition
 *       notes:          a line of data from the input file is placed
 *                       in the line vector. lineptr is left pointing
 *                       to the beginning of the line.
 */

getline() {
for (lineptr= line; (*lineptr= putchar(getc(iobuf))) != -1  &&
        *lineptr != 'nn'; lineptr++);   /* input, print, & store input data */

if (*lineptr == -1) {   /* eof condition? */
        if (lineptr != line)
                error(9);
        return(-1);
        }
lineptr= line;
return(0);
}
/*
 *                       S K I P S P A C E
 *
 *       arguements:     none
 *       returns:        0 if all's cool
 *                       -1 on an eof
 *       notes:          lineptr is left pointing to the first char not
 *                       a nn , tab, or a space.
 */

skipspace() {
while ( 1 ) switch( *lineptr ) {
        case SPACE:
        case TAB:
                lineptr++;
                break;
        case 'nn':
                if (getline() == -1) return(-1);
                break;
        default:
                return(0);
        }
}
```

```
/*
*                       G E T F I E L D
*
*       arguements:     1) target address for unit name
*                       2) target address for field name
*                       3) target address for field displacement
*       returns:        1 of all's cool
*                       0 if an error was encountered
*                       -1 if an eof was encountered
*       notes:          the expected format is [unitname] [.] [fieldname]
*                       if the format does not conform to that of the
*                       input stream, one of the format errors messages
*                       is printed.
*/

getfield (unitptr, fldptr, addr) char *unitptr, *fldptr; int *addr;  {
int n;

if ((n= getname(unitptr)) == -1) return(-1);      /* get unit name */
        else if (n == 0) {
                error(0);
                return(0);
                }

if ((n= gettok()) == -1) return(-1);              /* get '.' token */
        else if (n != '.') {
                error(0);
                return(0);
                }

if ((n= getname(fldptr)) == -1) return(-1);       /* get field name */
        else if (n == 0)  {
                error (0);
                return(0);
                }

if ((fldindex= findfield( fldptr )) == -1)  {     /* valid fld name? */
        error(1);
        return(0);
        }

*addr= 10 + displace[ fldindex ] *2 *FLDSIZE;
return(1);
}
/*
*                       E R R O R
*
*       arguements:     index number of the error message to print
*       returns:        zero
*       notes:          none.
*/

error(index) int index; {
int i:
if (sequence) {         /* error reporting flag? */
        for (i= 0; i < (lineptr- line- 1); i++) printf(" ");
        printf("^*n");
        printf("*ERROR: %s*n*n*n", errmesg[ index ]);
        errors++;
        }
}
/*
*                       O P E N F I L E
*
*       arguements:     target address for filename
*       returns:        zero
*       notes:          routine inputs a file name from the terminal
*                       and attempts to access it.
*/

openfile(s) char *s; {
int l;
char *ptr;

for (l= -1; ptr == s  ||  l == -1;) {
        printf("mnInput file? ");
        ptr= s;
        while ((*ptr= getchar()) != 'mn') ptr++;
        *ptr= 'm0';
        if ((l= fopen(s, iobuf)) == -1)
                printf("*ERROR: %s bad file namemn",s);
        }

printf("mnmn");
}
/*
*                       F I N D N A M E
*
*       arguements:     address of the name to be found
*       returns:        an ascii value of the first letter of the
*                               unit-type (a, c, m, p, v, or b) if
*                               the name was found
*                       zero if the name was not found
*       notes:          in the case where the name was found, the global
*                       variables unitindex and baseaddr are set.
*/


findname (s1) char *s1; {
int i,n;
for (i= 0; i < NUMUNITS; i++) {
        n= 0;
        switch (i) {
                case 0:
                        for (aluptr= alu; aluptr < alutoptr; aluptr++) {
                                if (compare( s1, aluptr)) {
                                        unitindex= n;
                                        baseaddr= aluptr;
                                        return('a');
                                        }
                                n++;
                                }
                        break;
                case 1:
                        for (memptr= mem; memptr < memtoptr; memptr++) {
                                if (compare( s1, memptr)) {
                                        unitindex= n;
                                        baseaddr= memptr;
                                        return('m');
                                        }
                                n++;
                                }
                        break;
                case  2:
                        for (cpuptr= cpu; cpuptr < cputoptr; cpuptr++) {
                                if (compare( s1, cpuptr)) {
                                        unitindex= n;
                                        baseaddr= cpuptr;
                                        return('c');
                                        }
                                n++;
                                }
                        break;
                case 3:
                        for (busptr= bus; busptr < bustoptr; busptr++) {
                                if (compare( s1,busptr)) {
                                        unitindex= n;
                                        baseaddr= busptr;
                                        return('b');
                                        }
                                n++;
                                }
                        break;
                case 4:
                        for (vsdptr= vsd; vsdptr < vsdtoptr; vsdptr++) {
                                if (compare( s1, vsdptr)) {
                                        unitindex= n;
                                        baseaddr= vsdptr;
                                        return('v');
                                        }
                                n++;
                                }
                        break;
                case 5:
                        for (perptr= per; perptr < pertoptr; perptr++) {
                                if (compare( s1, perptr)) {
                                        unitindex= n;
                                        baseaddr= perptr;
                                        return('p');
                                        }
                                n++;
                                }
                        break;
                };
        }
return(0);
}
/*
*                       E N T E R U N I T
*
*       arguements:     1) address of name to be entered
*                       2) unit type (a, b, c, m, p, or v)
*       returns:        unit type
*       notes:          use the subroutine to enter a new unit into
*                       a structure.
*/
enterunit (s1, s2) char *s1, *s2; {
switch (*s2) {
        case 'a':
                if (topindex[0] == MAXALU) return(0);  /* unit quota */
                baseaddr= alutoptr;
                stringxfer( s1, alutoptr++);    /* enter name field */
                unitindex= topindex[0]++;
                return('a');
        case 'm':
                if (topindex[1] == MAXMEM) return(0);
                memtoptr->pdpbase= &memory[ topindex[1]* 2* MEMSIZE ];
                baseaddr= memtoptr;
                stringxfer( s1, memtoptr++);
                unitindex= topindex[1]++;
                return('m');
        case 'c':
                if (topindex[2] == MAXCPU) return(0);
                baseaddr= cputoptr;
                stringxfer( s1, cputoptr++);
                unitindex= topindex[2]++;
                return('c');
        case 'b':
                if (topindex[3] == MAXBUS) return(0);
                baseaddr= bustoptr;
                stringxfer( s1, bustoptr++);
                unitindex= topindex[3]++;
                return('b');
        case 'v':
                if (topindex[4] == MAXVSD) return(0);
                baseaddr= vsdtoptr;
                stringxfer( s1, vsdtoptr++);
                unitindex= topindex[4]++;
                return('v');
        case 'p':
                if (topindex[5] == MAXPER) return(0);
                baseaddr= pertoptr;
                stringxfer (s1, pertoptr++);
                unitindex= topindex[5]++;
                return('p');
        default:
                return (0);
        }
}
/*
*                       C O N N E C T E D
*
*       arguements:     field type (1= input, 2=output, 3=io)
*       returns:        1 if field has been connected
*                       0 if field has no connections (god forbid)
*/

connected (fldtyp) int fldtyp; {
int *ptr;

switch( fldtype[ fldisp ] ) {
        case 1: return(match(baseaddr));        /* input field */
        case 2: return(*(ptr= baseaddr + POINTER)); /* output field */
        case 3: if (*(ptr= baseaddr + POINTER)) return(1);  /* io field */
                return(match( baseaddr ));
        }
}
/*
*       SMALL OR STOCK SUBROUTINES
*/

match(s) int s; {       /* looks for s in arc (looks for connection) */
int *ptr;               /* returns 0 if none, 1 if present */

ptr= arc;
ptr=+2;
while (ptr < arcptr) {
        if (*ptr++ == s) return(1);
                else if (*ptr == 0) ptr=+ 3;
                        else ptr=+ 2;
        }
return(0);
}

stringxfer ( s1, s2) char *s1, *s2; {   /* (s1) -> (s2) */
while ((*s2= *s1++) != '\0')
        s2++;
}

findfield (s1) char *s1; {      /* looks for (s1) in field table */
int i;
for (i= 0; fields[i] != 0; i++)
        if ( compare( fields[i], s1)) return(i);
```

```
return(-1);
}


compare( s1, s2)                    /* compare (s1) with (s2) */
char *s1, *s2; {            /* returns 1 if same, 0 if differ */
        while( *s1++ == *s2)
                if( *s2++ == '\0')
                        return(1);
        return(0);
}
/*

                    ****** MAIN2 ******

        arguments:      none
        returns:        1 if everything is OK
                        -1 if uncorrectable error
        notes:          mainline program for phase 2 of program

*/

main2()
{       int i, j, k, l, m, n;

        printf("\n\n\n\n                PHASE TWO:  ASSEMBLER \n\n");

        sequence = 1;
        initq();
        printq(head);
        initfaults();
        /* get name of memory where program will be loaded, check
           validity (i.e. if it is a memory unit */
        printf("\n\n\nName of memory where program will be loaded?\n");
        getmemname();
        count = 0;
        while ( ++count <= 3)
        {       if (findname(memload) == 'm') break; /* all's OK */
                printf("\n\nInvalid memory name.  Retype memory name\n\n");
                getmemname():
        }
        if( count > 3)  /* no correct memory name in 3 tries */
        {       printf("Check memory names from phase I \n",
                        "****** PROGRAM TERMINATED *****\n");
                return(-1);
        }
        /* initialize program address to beginning of given memory
           and initialize counting variables, symtab entries */
        progaddr = mem[unitindex].pdpbase;
        progbeg = progaddr;
        symptr = symtab[0];
        asymptr = symaddr;
        memstart = progaddr;
        nameptr = name;

        p2adptr = p2addr;

        /*              ***** PASS 1 *****                      */

        openfile(&phase2file);
        while(  (getline() != -1 ) )
        {
                memcount = progaddr - memstart;
                if  ( (temp = getname(name) ) <= 0)
                {       error(22);
                        goto p1loopbot; /* skip rest of line, go
```

```
                                to bottom of while loop*/
        }
        if (*lineptr == ':')  /* is a label */
        {       copy(name, symptr);
                symptr =+ 9;
                *symptr = 0;
                *asymptr++ = memcount;
                lineptr++;
                if ( getname(name) <= 0 )
                {       error(22);
                        goto p1loopbot;
                }
        }

        /*  name should now be an instruction */
        if(*nameptr == '$') /* PSEUDOS */
        {       if((temp=findstr(pseudos,&name[1]))<0) error(23);
                else
                {       switch(temp)
                        {
                        case 0:  /* DEF */
                                pseudo(10);
                                break;
                        case 1:  /* DEFO */
                                pseudo(8);
                                break;
                        }
                }
        }
        else
        if (*nameptr == '*')  /* fault injection */
        {       *progaddr =| 0100000;
                codefltinj();
        }
        else
        {       /* regular (non-fault injection) instruction*/
                if( (instnum = findstr(inst,name) )
                        > 0) codeinst(instnum);
                else /* can't find instruction */
                        error(23);
        }

                p1loopbot:   ;
}
symend = symptr;
symptr = symtab;
asymptr = symaddr;
/* print symbol table and associated address */
printf("                SYMBOL TABLE\n\n");
printf("      label            address\n\n");
while(symptr <  symend)
{
        printf("     %s              %6o\n", symptr,*asymptr);
        symptr =+ 9;
        asymptr++;
}


/*              ***** PASS 2 *****              */

printf("beg=%6o  end=%6o \n\n", symtab, p2adptr);
p2endaddr = p2adptr;
p2adptr = p2addr;
openfile(&phase2file);
while( getline() != -1)
{
```

```
                if (getname(name) <= 0) goto p2loopbot;
                if(*lineptr == ':')
                {       lineptr++;
                        if (getname(name) <= 0) goto p2loopbot;
                }
                /* name should be an instruction. If the instruction
                   is a branch, processing is required, o/w get
                        next instruction */
                if( (instnum=findstr(p2inst, name)) >= 0)
                {       /* is a branch or memory instruction */
                        if ( instnum==2 || instnum == 3)
                        {       /*conditional--get condition code*/
                                if(spaces()<=0)goto p2loopbot;
                                if((cond=convert())<=0)
                                {       error(29);
                                        goto p2loopbot;
                                }
                                if(getcomma(0) <= 0) return;
                        }
                        if ( spaces() <= 0) goto p2loopbot;
                        if( instnum >= 6) /* memory reference */
                        {       if(getfldid()<=0)
                                        goto p2loopbot;
                                if(getcomma(0) <= 0) return;
                        }
                        /* get label and locate it in symbol table */
                        if (getname(name) <= 0)
                        {       error(22);
                                goto p2loopbot;
                        }
                        if ( (instnum=labelfind()) < 0)
                        {
                                /* label not in symtab */
                                error(25);
                        }
                        else
                        {       while( *lineptr==' ' ||
                                    *lineptr == 'mt') lineptr++;
                                if(*lineptr != 'mn')
                                {       n = convert();
                                        **p2adptr++ =| symaddr[instnum]
                                                                + n;
                                }
                                else
                                        **p2adptr++ =| symaddr[instnum];
                        }
                }
                p2loopbot:  ;
        }
        if (p2adptr != p2endaddr) printf("assembler errormn");
        if (errors > 0) return(-1);
        return(1);
}
/*

        ******                  FINDSTR                 ******

        arguments:  arr, str where str is the address of a
                    string to be searched for in array beginning
                    at address arr.
*/

findstr (arr, str) /* search for string str in array arr */
    char *arr[ ];
    char *str;  {
        int i,j,m;
        for (i=0; arr[i] != 0; i++) {
                for (j=0; (r = arr[i][j]) == str[j]  &&
                                r != '\0'; j++);
                if (r == str[j])
                        return (i);
        }
        return (-1);
}


/*
                    ***** LABELFIND *****

                arguments:      none
        returns:        row number in symtab where label is found
                                if the label was found in symtab
                        -1 if the label was not found
        notes:          searches for name (that character string)
                        in symbol table

*/


labelfind()
{    char r;
     int i,j;
        for( i=0; symtab[i][0] != 0; i++)
        {
                for(j=0; (r=symtab[i][j]) == name[j] &&
                                r != '\0'; j++ );
                if ( r == name[j]) return(i);
        }
        return(-1);
}
/*                   ***** PSEUDOS *****

        arguments:      the base which the numbers are to be
                        interpreted
        returns:        none
        notes:          gets ascii characters from input
                        and converts them to a number
*/
pseudo(base)
int base;
{       int n, sign;
        while( 1 )
        {       if( (temp=spaces()) < 0)
                {       error(13);
                        return;
                }
                if(*lineptr == '-')
                {       sign = -1;
                        lineptr++;
                        spaces();
                }
                else sign = 1;
                temp = lineptr;
                n = 0;
                if ( base == 10)
                {       while (*lineptr >= '0' && *lineptr <= '9')
                                n = n*10 + *lineptr++ - '0';
                }
                else
                {       while(*lineptr >= '0' && *lineptr <= '7')
                                n = n*8 + *lineptr++ - '0';
                }
                if (temp == lineptr)
                {       error(15);
                        return (-1);
                }
                *progaddr++ = n * sign;
                if (spaces() == 0) break;
                if( getcomma(0) < 0) break;
        }
}




/*                      ***** CODEFLTINJ *****

        arguments:      none
        returns:        none
        notes:          calls appropriate routines for special
                        fault injection instructions
*/
codefltinj()
{
        /* check for special fault injection */
        if ( (instnum = findstr(sfinst,&name[0]) )
                >= 0) /* is special fault injection */
                specfault(instnum);
        else
        if( (instnum = findstr(inst,&name[1]) )
                >= 0 )  codeinst(instnum);
        else /*can't find instruction */
                error(23);
}

/*
                ****** SPACES ******

        arguments:  none
        returns:    -1 if end of line is encountered
                     0 if semicolon is encountered
                     1 if any other character is encountered
        notes:       skips spaces until hits character other
                     than blank and returns  value corresponding
                     to last character

*/
spaces()
{
        while (*lineptr == ' ' || *lineptr == '¢t') lineptr++;
        if (*lineptr == '¢n') return ( -1);
        if (*lineptr == ';' ) return (  0);
        return (1);
}
/*

                ****** BITCONVERT ******

        arguments:      none
        returns:        -1 if a number is not found or the
                            number is < 0 or > 31
                        the decimal value if conversion was OK and
                            result was valid
        notes:          converts ascii character pointed to by
                        lineptr to its decimal value, and moves
                        lineptr to first non-numeric character
                        encountered and checks the value as above
```

```
*/

bitconvert()
{
    int n;
        temp = lineptr;
        n=0;
        while (*lineptr >= '0'  && *lineptr <= '9')
                n = n * 10 + *lineptr++ - '0';
        if (temp == lineptr || n > 31)
        {       error(13);
                return( -1);
        }
        return(n);
}
/*
                ****** CONVERT *******

        arguments:      none
        returns:        0 if a number is not found
                        the decimal value if conversion was OK
        notes:          converts ascii character pointed to by
                        lineptr to its decimal value, and moves
                        lineptr to first non-numeric character
                        encountered, checking to insure number
                        is > 0

*/
convert()
{
    int n, sign;
        if (*lineptr == '-') sign = -1;
        else sign = 1;
        if( *lineptr == '+' || *lineptr == '-')
        {       lineptr++;
                spaces();
        }

        temp = lineptr;
        n = 0;
        while( *lineptr >= '0' && *lineptr <= '9')
                n = n * 10 + *lineptr++ - '0';
        if( temp == lineptr)
        {       error(15);
                return (-1);
        }
        n = n * sign;
        return(n);
}

/*

                ****** GETSA0 ******

        arguments:      address where 32 bit stuck at fault mask
                        will be placed
        returns:        0 if last character processed is ';'
                        -1 if last character processed is new line
                        1 if not one of the above--problem code
        notes:          gets ascii values of bit numbers to be
                        stuck at 0 and generates the appropriate
                        mask placed in *addr and *(addr + 1)
*/

getsa0(addr)
```

```
int *addr;
{
        *addr = 0177777;  /* initialize to all ones */
        *(addr + 1) = 0177777;
        while (spaces() > 0)
        {
                if ( (temp = bitconvert() ) >= 0)
                {
                        if (temp < 16) *addr =& zeroes[temp];
                        else *(addr + 1) =& zeroes[temp - 16];
                }
                else return(1); /* bad number */
                if ( (temp = spaces() )  <= 0) return(temp);
                if (*lineptr != ',') error(11);
                lineptr++;
        }
        error(12);
        return(1);
}
/*

                ****** GETSA1 ******

        arguments:      address where 32 bit stuck at fault mask
                        will be placed
        returns:        0 if last character processed is ';'
                        -1 if last character processed is new line
                        1 if not one of the above--problem code
        notes:          gets ascii values of bit numbers to be
                        stuck at 0 and generates the appropriate
                        mask placed in *addr and *(addr + 1).
*/

getsa1(addr)
int *addr;
{
        *addr = 0;  /* initialize to all ones */
        *(addr + 1) = 0;
        while (spaces() > 0)
        {
                if ( (temp = bitconvert() ) >= 0)
                {
                        if (temp < 16) *addr =| ones[temp];
                        else *(addr + 1) =| ones[temp - 16];
                }
                else return(1); /* bad number */
                if ( (temp = spaces() )  <= 0) return(temp);
                if (*lineptr != ',') error(11);
                lineptr++;
        }
        error(12);
        return(1);
}
/*
                ****** FREQVALUE ******

        arguments:      address where the two integer values
                        will be placed
        returns:        none.
        notes:          gets ascii values of integers used in
                        calculation of frequency of fault
                        injection (random stuck at's)
                        and places the numerator in addr and the
                        denominator in  addr + 1
*/
```

```
freqvalue(addr)
int *addr;
{
        if ( spaces() < 0 )
        {       /* unexpected end of line */
                error(16);
                return;
        }
        if( *lineptr++ != ';') return;
        if ( spaces() <= 0)
        {       error(16);
                return;
        }
        if ( (temp = convert() )  <= 0 ) /* get numerator */
        {       /* invalid number encountered */
                error(15);
                return;
        }
        *addr++ = temp;
        if (spaces() <= 0)
        {       /* eof or ; encountered */
                error(16);
                return;
        }
        if ( *lineptr++ != '/') error(12);
        if (spaces() <= 0)
        {       error(15);
                return;
        }
        if( (temp = convert() ) <= 0) /* get denominator */
        {       /* invalid number encountered */
                error(15);
                return;
        }
        *addr++ = temp;
}
/*

        ******          GETFLDID                ******

        arguments:      none
        returns:        -1 if an error in the field is encountered
                          or an eof is encountered
                        0 if the unit name cannot be found
                        1 if everything is OK
        notes:          this routine gets the field ID number
                         of the string which begins at the address
                        lineptr.  At the end of the routine, the
                        id can be found in
                        "*(baseaddr + fldisp + fldid)"
*/

getfldid()
{
        if ( (temp = getfield(&unitname,&fieldname,&fldisp) )  < 1)
                /* check for error or eof */
        {
                if (temp == 0) error(0);  /*illegal field name */
                else error(9);
                return(-1);
        }
        if ( findname (&unitname) == 0)   /*  can't find unitname */
        {
                error (10);
                return(0);
        }
        return(1);
}
/*

                ***** SPECFAULT *****

        arguments:      number of row in array sfirst the instruction
                        mnemonic is found
        returns:        none.
        notes:          routine to generate code for special fault
                        injection instructions.

*/
specfault(num)
int num;
{
        *progaddr =| 0170000; /*insert code indicating inst is a
                                special fault injection instruction
                                (in bits 0-3)     */
        *(progaddr + 1) =| faultcode [num]; /* set bits 16-18 to
                        appropriate instruction code */

        /* get field id number determined in phase I, place in
           bits 4-15    */
        if( getfldid() <= 0) return; /* bad field */
        *progaddr =| *(spare=baseaddr + fldisp + FLDID);
        progaddr =+ 2;
        if( num == 0) /* RAF */ return;
        if ( getcomma(0) <= 0) return;  /* bad format */

        /* generate remaining operands if present  */
        Switch (num) {

        case(1):  /* DZ */
                if ( (temp=convert()) < 0)  error(27);
                else
                {       *progaddr = temp;
                        progaddr =+ 2;
                }
                break;
        case(2):  /*  *SA0  */
                getsa0(progaddr);
                progaddr =+ 2;
                break;
        case(3):  /*  *SA1 */
                getsa1(progaddr);
                progaddr =+ 2;
                break;
        case(4):  /*  *PSA0  */
                getsa0(progaddr);
                progaddr =+ 2;
                freqvalue (progaddr);
                progaddr =+ 2;
                break;
        case(5):  /*  *PSA1  */
                getsa1(progaddr);
                progaddr =+ 2;
                freqvalue (progaddr);
                progaddr =+ 2;
                break;
        case(6):  /*  *PF0  */
                getsa1(progaddr);
                progaddr  =+ 2;
                break;
        case(7):  /* *PF1 */
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
                getsa0(progaddr);
                progaddr =+ 2;
                break;
        }
}
/*

                      ******GETCOMMA******

        arguments:      -1 if comma optional
                        0 if comma required
        returns:        -1 if error encountered
                        0 if optional comma is not present
                        1 if all is OK
        notes:          skips spaces until finds a comma
                        and then skips spaces, leaving lineptr
                        pointing to the first non-blank character
                        which is not an end of line or ';'

*/

getcomma(opt)
int opt;
{
        if (spaces() <= 0)
        {       if (opt == 1) return(0);/* optional comma not present*/
                error(18);
                return(-1);
        }
        if (*lineptr++ != ',')
        {       error(17);
                return(-1);
        }
        if (spaces() <= 0)
        {       error(18);
                return(-1);
        }
        return(1);
}



/*                     ***** LABOFFSET *****

        arguments:      none
        returns:        -1 if error
                        0 otherwise
        notes:          checks to see if there is an offset following
                        the label (i.e., plus or minus a constant)

*/
laboffset()
{       if (spaces() <= 0) return(0);
        if ( *lineptr == ',') return(0);
        if(*lineptr != '+' && *lineptr != '-')
        {       error(25);
                return(-1);
        }
        lineptr++;
        if(spaces() <= 0) return(-1);
        convert();
}
/*

                      ****** IMM_OP ******
        arguments:      row number of array inst where the instruction
                        is found
        returns:        none
        notes:          routine to generate code for 2-operand
                        instructions

*/

imm_op(inum)
int inum;
{
        /* assign proper bit values to bits 1-3 */
        *(progaddr) =| 030000;
        /* set bits 16-19 */
        *(progaddr + 1) =| regcode[inum];

        /* get field id number and place in bits 4-15 */
        if ( getfldid() <= 0) return;  /* bad name, skip rest of line*/

                *progaddr++ =| *(spare=baseaddr + fldisp + FLDID);

        /* get immediate value */
        if ( getcomma(0) <= 0) return;  /* bad format */
        if( (temp=convert() ) < 0)
        {       *progaddr =| ones[4]; /* set sign bit */
                temp = abs(temp);
        }
        if (temp >= 4096)
        {       /* number is too large */
                progaddr++;
                error(21);
                return;
        }
        if (temp != 0) *progaddr =| temp;
        progaddr++;
}
/*

                        ****** TWO_OP ******

        arguments:      row number of array inst where the
                        instruction mnemonic is found
        returns:        none
        notes:          routine to generate code for 2-operand
                        instructions

*/

two_op(inum)
int inum;
{       /* bits 1-3 are all zeroes initially as desired
           set bits 16-19 to specific instruction type */
        *(progaddr + 1) =| regcode[inum];

        /* get field id numbers and place in bits 4-15 and
                20-31 respectively */
        if (getfldid() <= 0) return; /* error in getting field */
        *progaddr =| *(spare=baseaddr + fldisp + FLDID);
        if (getcomma(0) <= 0) return;  /* bad format*/
        if( inum <= 8) /* no memory reference-get second field*/
        {
                if(getfldid() <= 0) return;
                *(progaddr + 1) =| *(spare=baseaddr + fldisp + FLDID);
        }
        else
        {       /* LDM or STM */
                *p2adptr++ = progaddr + 1;
                if( getname(name) <= 0) return;
                laboffset();
        }

        /* check for indirection */
        if( getcomma(1) <= 0) /* no indirection */
        {       progaddr =+ 2;
                return;
        }
        if (*lineptr != 'I')
        {       error(19);
                return;
        }
        /* set bit one to indicate indirection */
        *(progaddr) =| ones[1];
        progaddr =+ 2;
}
/*

                    ****** BRANCH_OP ******

        arguments:      row number of array inst where instruction
                        mnemonic is found
        returns:        none
        notes:          routine to generate code for branch and halt
                        instructions

*/

branch_op(inum)
int inum;
{   int n;
        /* assign proper bit values to bits 1-3 */
        *progaddr =| 050000;
        /* set bits 16-19 */
        *(progaddr + 1) =| regcode[inum];
        if (inum >= 24   &&  inum <= 26 ) /* a branch instruction*/
        {       if (spaces() <= 0)
                {       error(18);
                        return;
                }
                if( inum == 25) /* conditional-get cond. code*/
                {       if( (n=convert()) > 31  || n < 0 )
                        {       error(23);
                                return;
                        }
                        *(progaddr+1) =| n;
                        if(getcomma(0) < 0) return;
                }
                /* get label */
                if ( getname(name) <= 0) return;
                if (laboffset() < 0) return;
                if(getcomma(1) == 1)
                {       /*should be indirect */
                        if(*lineptr == 'I')
                                *(progaddr + 1) =| 010000;
                        else error(19);
                }
                *p2adptr++ = progaddr;
        }
        /*  HLT or RET requires no further arguments */
        progaddr =+ 2;
}
/*
```

```
                    ***** OTHER_OP *****

        arguments:      row number of array inst where the
                        instruction mnemonic is found
        returns:        none
        notes:          routine to generate code for various
                        instructions

*/
other_op(inum)
int inum;
{
        /* assign proper bit values to bits 1-3 */
        *progaddr =| 060000;
        /* set bits 16-19 */
        *(progaddr + 1) =| regcode[inum];
        *(progaddr + 1) =| 5;

        switch(inum)
        {
        case(48):
                if( spacecheck() > 0)   /* proper format */
                {
                        switch (*lineptr)
                        {       case('I'): break;
                                case('O'): *(progaddr + 1) =| ones[3];
                                           break;
                                default: error(20);
                        }
                        lineptr++;
                        if(getcomma(0) <=0) return;
                        if(getname(name) <= 0)
                        {
                                error(25);
                                return;
                        }
                        if( (c=findname(name)) != 'p')
                        {
                                error(26);
                                return;
                        }
                        *progaddr =| *(spare=baseaddr + 10 + FLDID);
                }
                else  error(20);
                break;
        case(49):
                /* BOMB */
                break;
        }
        progaddr =+ 2;
}
/*

                    ****** SPACECHECK ******

        arguments:      none
        returns:        -1 if eof or ;
                        1 if all is well
        notes:          skips to first nonblank character and
                        checks for ; or eof

*/
spacecheck()
{       if (spaces() <= 0)
        {
                error(13);
```

```
                        return(-1);
                }
                return(1);
}
/*

                                ****** ONE_OP *******

                arguments:      row number of array inst where the
                                instruction is found
                returns:        none
                notes:          routine to gener ate code for one oper and
                                instructions

*/
one_op(inum)
int inum;
{
                /* assign proper bit values to bits 1-3 */
                *progaddr =| 020000;
                /* set bits 16-18 */
                *(progaddr + 1) =| regcode[inum];

                /* get field id number, place in bits 4-15 */
                if( spaces() <= 0 )
                {       error(18);
                        return;
                }
                if ( getfldid() <= 0) return;  /* bad field name */
                *progaddr =| *(spare=baseaddr + fldisp + FLDID );
                progaddr =+ 2;
}
/*

                                ****** UNIT_OP ******

                arguments:      row number of array inst where the
                                instruction is found
                returns:        none
                notes:          routines to generate code for 2-operand
                                instructions

*/

unit_op(inum)
int inum;
{
        int i;
                /* assign bits 1-3 proper value */
                *progaddr =| 010000;
                /* set bits 16-19 */
                *(progaddr + 1) =! regcode[inum];

                /* get unit id number place in bits 4-15 */
                if( getname(name) <= 0)
                {       error(25);
                        return;
                }
                if ( (c=findname(name) ) == 0)
                {       error(25);
                        return;
                }
                *progaddr =! *(spare = baseaddr + 10 + FLDID);
                if( inum <= 41)
                {       progaddr =+ 2;
                        if (c != 'a') error(25);
                        return;
                }
                else
                {       switch(inum)
                        {
                        case(42):  /* CLU */
                                if( c != 'a' && c != 'c')  error(26);
                        case(43):  /* UDUMP */
                                settype();
                                progaddr =+ 2;
                                break;
                        case(44):  /* MDUMP */
                                *(progaddr + 1) =| 01;
                                if ( c != 'm') error(26);
                                progaddr =+ 2;
                                for(i=1; i<=2; i++)
                                {       /* get memory boundaries */
                                        if(getcomma(0) <= 0)
                                        {       error(24);
                                                return;
                                        }
                                        else
                                        if((temp == 0) && *(lineptr-1) != '0')
                                        {       error(24);
                                                return;
                                        }
                                        *progaddr++ = temp;
                                }
                                break;
                        case 45:  /* VOTE */
                                *(progaddr + 1) =| 04;
                                if ( c != 'v') error(26);
                                progaddr =+ 2;
                                break;
                        case 46:  /* SST */
                        case 47:  /* RST */
                                settype();
                                progaddr =+ 2;
                                break;
                        }
                }
        }
        /*                      ***** SETTYPE *****
                arguments:      none
                returns:        none
                notes:          called by unitq, uses the last twelve
                                bits of *(progaddr + 1) i.e. opnd2,
                                to indicate unit type (alu, cpu, etc.)
        */
        settype()
        {
                switch( c )
                {
                case 'm':
                        *(progaddr + 1) =| 01;
                        break;
                case 'c':
                        *(progaddr + 1) =| 02;
                        break;
                case 'b':
                        *(progaddr + 1) =| 03;
                        break;
                case 'v':
                        *(progaddr + 1)  =| 04;
                        break;
                case 'p':
```

```
                *(progaddr + 1) =| 05;
                break;
        }
}
/*

                    ****** CODEINST ******

        arguments:      row number of array inst where the instruction
                        mnemonic is found
        returns:        none
        notes:          routine to generate code for "regular"
                        instructions and asembly type fault
                        injection instructions

*/
codeinst(inum)
int inum;
{
        if (inum >= 0 && inum <= 10)
                two_op(inum);
        else
        if (inum >= 11 && inum <= 23)
                imm_op(inum);
        else
        if ( inum >= 24 && inum <= 28 )
                branch_op(inum);
        else
        if (inum >= 29 && inum <= 34 )
                one_op(inum);
        else
        if (inum >= 35 && inum <= 47)
                unit_op(inum);
        else
                other_op(inum);
}
/*

                ****** GETMEMNAME ******

        arguments:      none
        returns:        none
        notes:          reads name of memory where program will be
                        loaded and puts name in memload
*/

getmemname()
{
    int letcount;  /* counts letters in memload to insure
                                there are <= 8 letters   */
        mlptr = memload;
        while(  (c = getchar() )  == ' ');
        *mlptr++ = c;
        letcount = 1;
        while (  (c=getchar() )  != ' '   &&  c != '\n'
                        && letcount++ <= 8)
                *mlptr++ = c;
        *mlptr = '\0';
}
/*
                    ***** MAIN3 ******

        arguments:      none
        returns:        none
        notes:          decodes instructions, places events on event
```

```
                                queue, and executes events

*/
main3()
{
        time(tvec);
        srand(tvec[1]);
        dummy = 0;
        lz = dummy;
        dummy = 1;
        longone = dummy;
        clock = 0;
        pa = progbeg;
        printq(head);
        while( pa < progend)
        {
                xevents(clock);
                if( pa >= progend)  /* check if halt */ break;
                ginst();
                if( faultbit == 0)  clock =+ DECODETIME;
        }
        fintime = 32750;
        fintime =* fintime;
        xevents(fintime);
}
/*
                ****** FLDADDR *****

        arguments:      12 bit operand field
        returns:        returns the address value as an integer
        notes:          generates unit type, unit index, and
                        field index from the 12 bit operand
*/

fldaddr(opnd)
int opnd;
{       char *addr;
        unittype = opnd >> 9;
        unitindex = opnd >> 4  & 037;
        fldindex = opnd & 017;

        switch(unittype)
        {
                case 0:
                        addr = &alu[unitindex];
                        break;
                case 1:
                        addr = &mem[unitindex];
                        break;
                case 2:
                        addr = &cpu[unitindex];
                        break;
                case 3:
                        addr = &bus[unitindex];
                        break;
                case 4:
                        addr = &vsd[unitindex];
                        break;
                case 5:
                        addr = &per[unitindex];
                        break;
        }
        addr =+ 10 + fldindex * 2 * FLDSIZE;
        return(addr);
}
/*
```

```
                    *****PARSEWORD*****

        arguments:      none
        returns:        none
        notes:          parses the instruction pointed to by pa
*/
parseword()
{
        faultbit = *pa >> 15 & 01;
        optype = *pa >> 12  & 07;
        opnd1 = *pa & 07777;
        opnd2 = *(pa + 1) & 07777;
        opspec = *(pa + 1) >> 12 & 017;
}
/*
                    ***** CREATENODE *****

        arguments:      none
        returns:        the "created" node if all is OK
                        0 if all nodes have been used
        notes:          finds a node not in use and returns a
                        pointer to it
*/
struct node *createnode()
{
 register struct node *nptr;
 register struct node *zptr;

        nptr = &nodes[0];
        zptr = &nodes[NUMNODES]:

        while (nptr < zptr) {
                if (nptr->used == 0) {
                        (nptr->used)++;
                        return( nptr );
                }
                nptr++;
        }
        printf("no free nodes--increase NUMNODES in define file\n");
        exit();
}

/*
                ***** FREE *****

        arguments:      pointer to a structure
        returns:        none
        notes:          resets the used field of qptr to zero
                        to indicate node is availble for use
 */
free( qptr ) struct node *qptr;
{
        (qptr->used)--;
}
/*
                    ***** INSERT *****

        arguments:      two pointers to structures (see notes)
        returns:        none
        notes:          insert node 'newnode' before 'qnode',
                        a node on the event queue
*/

insert(qnode, newnode)
    struct node *qnode,
                *newnode;
```

```
{    struct node *temptr;
        temptr = qnode->prev;
        newnode->prev = temptr;
        newnode->next = qnode;
        qnode->prev = newnode;
        temptr->next = newnode;
}
/*
                    ***** INITQ *****

        arguments        none
        returns:         none
        notes:           creates the initial event queue with the
                         header node and end node. The header node
                         time value is -1, its pointer prev is 0.
                         The clock value of the end node is 32,767,
                         its pointer next is 0;
*/

initq()
{    struct node *new;
        head = &headnode;
        head->time = -1;
        new = createnode();
        dummy = 32760;
        new->time = dummy;
        new->time =* dummy;
        head->next = new;
        new->prev = head;
        head->prev = 0;
        new->next = 0;

        p = head->next;
}



/*                   ***** QINST *****

        arguments:       none
        returns:         none
        notes:           queues the instruction in pointed
                         to by global address pa
*/
qinst()
{
        parseword();
        /* place instruction on event queue */
        switch (optype)
        {
        case 0:  /* Two operand instruction*/
                twoq(0);
                pa =+ 2;
                break;
        case 1:  /*  Unit instruction */
                unitq();
                pa =+ 2;
                break;
        case 2:  /*  Single operand */
                oneq();
                pa =+ 2;
                break;
        case 3:  /*  Immediate operand */
                immq();
                pa =+ 2;
                break;
        case 4:  /*  Two operand indirect */
                twoq(1);
                pa =+ 2;
                break;
        case 5:  /* Branch */
                branchq();
                break;
        case 6:
                otherq();
                pa =+ 2;
                break;
        case 7:  /* Special fault injection */
                faultq();
                pa =+ 2;
                break;
        }
}
/*
                      *****QEVENT*****

        arguments:      all arguments needed to insert the event
                        specified on the event queue
        returns:        none
        notes:          places specified event on event queue
                        according to clocktime.  For equal clock
                        times, it is placed at the bottom of the list
                        for events of that clocktime
*/

qevent(tm,tp,sp,a1,a2,v,msk,fr)
    long
        tm; /* complete time */
    int
        tp, /* operation type */
        sp; /* operation specification */
    char
        *a1, /* address 1 */
        *a2; /* address 2 */
    int
        v; /* actual decimal value */
    long
        msk;  /* mask for stuck-at f.i. instructions */
    float
        fr; /*frequency for random stuck at f.i. instructions */
{
    struct node *new;
    long *lptr1, *lptr2;
    int *iptr;
        new = createnode();
        new->time = tm;
        new->type = tp;
        new->spec = sp;
        new->add1 = a1;
        new->add2 = a2;
        new->val = v;
        new->mask = msk;
        new->freq = fr;
        lptr1 = new->add1;
        lptr2 = new->add2;
        if(initialization) new->time = fltime;
        iptr = &new->mask;

        p = head -> next;
        while (p->time <= new->time) p = p->next;
        insert(p,new);
```

```
}
/*
                    ***** PRINTQ *****

        arguments:      pointer to node on queue
        returns:        none
        notes:          prints the values of the event queue
                        starting at qptr
*/

printq(qptr)
struct node *qptr;
{    struct node *qp;
    long *lptr1, *lptr2;
    long l1,l2;
    int *iptr;
        qp = qptr;
        printf(" time    type    opspec   add1    add2   value");
        printf("     mask       freq\n\n");
        while( qp-> next != NULL)
        {       lptr1 = qp->add1;
                lptr2 = qp->add2;
                l1 = *lptr1;
                l2 = *lptr2;
                iptr = &qp->mask;
                printf("%s  %3d  %3d  ",
                        locv(qp->time), qp->type, qp->spec);
                printf("%s  ",locv(l1) );
                printf("%s  %5d  %6o%6o  %f\n",locv(l2), qp->val,
                        iptr[0 ], iptr[ 1], qp->freq);
                qp = qp->next;
        }
}
/*
                    ***** TWOQ *****

        arguments:      ind = 1 => indirect, = 0 => direct
        returns:        none
        notes:          enters a two operand instruction on the event
                        queue.  If the instruction is not fault
                        injection, the associated operation time
                        plus decode time is added to the clock to get
                        event completion time.  For fault injection
                        instructions, completion time is the max
                        of the last calculated completion times of
                        of the two fields used.
*/

twoq(ind)
int ind:
{   long *addr1,  *addr2;
    char *cbase1, *cbase2;
   int itemp;
        /* check for indirection */
        if (ind == 1)
        {       opnd1 = *(spare = opnd1);
                opnd2 = *(spare = opnd2);
        }

        if (opspec >= 9)  /* memory reference instruction*/
        {
                cbase1 = fldaddr(opnd1);
                addr1 = cbase1 + CLOCK;
                cbase2 = progbeg + opnd2;
                if(faultbit)
                {
                        fintime = *addr1;
                        qevent(fintime,optype,opspec,cbase1,cbase2,0,lz,0.0);
                }
                else
                {
                        fintime=max(fintime,clock);
                        if(fintime <= clock + DECODETIME)
                                fintime =+ DECODETIME;
                        fintime =+ FETCHTIME;
                        qevent(fintime,optype,opspec,cbase1,cbase2,0,lz,0.0);
                }
                return;
        }
        cbase1 = fldaddr(opnd1);
        cbase2 = fldaddr(opnd2);
        addr1 = cbase1 + CLOCK;
        addr2 = cbase2 + CLOCK;
        fintime = max( *addr1, *addr2 );
        if (faultbit)
        {
                *addr1 = fintime;
                *addr2 = fintime;
                qevent(fintime,optype,opspec, cbase1, cbase2,0,lz,0.0);
        }
        else
        {
                fintime = max( fintime,clock);
                if ( fintime <= clock + DECODETIME)
                        fintime =+ DECODETIME;
                if(opspec <= 6) addoptime();
                else
                {
                        switch (opspec)
                        {
                        case 7:
                                fintime =+ MOVETIME;
                                break;
                        case 8:
                                fintime =+ CPRTIME;
                                break;
                        }
                }
                if( ! initialization)
                {
                        *addr1 = fintime;
                        *addr2 = fintime;
                }
                qevent(fintime, optype, opspec,cbase1,cbase2,0,lz,0.0);
        }
}
/*
                  ***** ONEQ *****

        arguments:      none
        returns:        none
        notes:          enters a one operand instruction on the
                        event queue.  If the instruction is not
                        fault injection, the associated operation
                        time plus decode time is added to the maximum
                        of the clock value & the last field completion
                        time to get event completion time. For fault
                        injection instructions, completion time is
                        the last calculated time of the field.

*/
```

```
oneq()
{    long *addr;
     int *iptr;
     char *cbase;
        cbase = fldaddr(opnd1) ;
        addr = cbase + CLOCK;
        if (faultbit)
        {
                fintime = *addr;
                qevent(fintime,optype,opspec,cbase,0,0,lz,0.0);
        }
        else
        {
                fintime = max( *addr, clock);
                if( fintime <= clock + DECODETIME)
                        fintime =+ DECODETIME;
                switch(opspec)
                {
                case 0:
                        fintime =+ NEGTIME;
                        break;
                case 1:
                        fintime =+ CPLTIME;
                        break;
                case 2:
                        fintime =+ SQRTIME;
                        break;
                case 3:
                        fintime =+ ABSTIME;
                        break;
                case 4:
                        fintime =+ CLRTIME;
                        break;
                case 5:
                        fintime =+ PARTIME;
                        break;
                }

                if( ! initialization) *addr = fintime;
                qevent(fintime,optype,opspec,cbase,0,0,lz,0.0);
        }
}
/*

                    ***** IMMQ *****

        arguments:      none
        returns:        none
        notes:          enters an immediate operand instruction
                        on the event queue.  If the instruction is
                        not fault injection, the associated operation
                        time plus decode time is added to the clokck
                        to get event completion time.  For fault
                        injection instructions, completion time is
                        the last calculated completion time of
                        te field
*/
immq()
{    long *addr;
    char *cbase;
    int *iptr;
        cbase = fldaddr(opnd1) ;
        addr = cbase + CLOCK;
        if (faultbit)
        {
                fintime = *addr;
                qevent(fintime, optype, opspec, cbase, 0,opnd2,lz,0.0);
        }
        else
        {
                fintime = max( *addr, clock);
                if( fintime <= clock + DECODETIME)
                        fintime =+ DECODETIME;
                if( opspec <= 6) addoptime();
                else
                {
                        switch(opspec)
                        {
                        case 7:
                                fintime =+ LOADTIME;
                                break;
                        case 8:
                                fintime =+ CPRTIME;
                                condtime= fintime;
                                break;
                        case 9:
                        case 10:
                                fintime =+ LSHTIME;
                                break;
                        case 11:
                        case 12:
                                fintime =+ ASHTIME;
                                break;
                        }
                }

                if( ! initialization) *addr = fintime;
                addr = cbase;
                qevent(fintime,optype,opspec,cbase, 0, opnd2,lz,0.0);
        }
}
/*
                        ***** BRANCHQ *****

        arguments:      none
        returns:        none
        notes:          enters a branch instruction on the
                        event queue
*/

branchq()
{
        if( (opspec & 01) == 1)  /* indirect */
        {
                opnd1 = *(spare = &opnd1);
        }
        lcond = opnd2;
        if( faultbit)
        {       qevent(clock,optype,opspec,0,0,opnd1,lcond,0.0);
                return;
        }
        if (opspec == 2 || opspec == 3)
                {
                        clock= condtime;
                        fintime= condtime;
                        pa =+ 2;
                }
        else
                fintime= clock+ DECODETIME;
        qevent(fintime,optype,opspec,0,0,opnd1,lcond,0.0);
}
/*                       ***** UNITQ *****
```

```
        arguments:      none
        returns:        none
        notes:          enters a unit operand instruction
                        on the event queue.  If the
                        instruction is not fault-injecting,
                        fintime= max( max(all unit field
                        times) , clock) + assoc. operation
                        time (+ DECODE if nec.).  For
                        fault-injection instructions,
                        completion time is the max of the
                        field times of the unit fields.
                        Each field within the unit is
                        then assigned the new time.
*/
unitq()
{       int i;
        long *addr;
        char *cbase, *ca;
           cbase = fldaddr(opnd1) ;
           /* examine completion times of each field within
                   the unit and compute the max */
           maxtime(cbase, numfields[opnd2]) ;
           fintime = mtime;

           if (faultbit)
           {
                   qevent(fintime,optype,opspec,cbase,0,opnd2,lz,0.0) ;
                   for (i=0; i<= (numfields[opnd2] -1) ;i++)
                   {
                           addr = cbase + CLOCK + (i * FLDSIZE * 2) ;
                           *addr = fintime;
                   }
           }
           else
           {
                   fintime = max(fintime, clock) ;
                   if( fintime <= clock + DECODETIME)
                           fintime =+ DECODETIME;
                   if (opspec <= 6) addoptime() ;
                   else
                   {       switch(opspec)
                           {
                           case 7:
                                   fintime =+ CLRTIME;
                                   break;
                           case 8:
                           case 9:
                                   fintime =+ DUMPTIME;
                                   break;
                           case 10:
                                   fintime =+ VOTETIME;
                                   break;
                           }
                   }
                   qevent(fintime,optype,opspec,cbase,0,opnd2,lz,0.0) ;
                   for (i=0; i<= (numfields[opnd2] -1) ;i++)
                   {
                           addr = cbase + CLOCK + (i * FLDSIZE * 2) ;
                           *addr = fintime;
                   }
           }
}
/*                 ****** ADDOPTIME *****

        arguments:      none
        returns:        none
        notes:          adds common operation times to fintime
*/
addoptime()
{
        switch (opspec)
        {
        case 0:
        case 1:
                fintime =+ ADDTIME;
                break;
        case 2:
                fintime =+ MULTIME:
                break;
        case 3:
                fintime =+ DIVTIME;
                break;
        case 4:
                fintime =+ ANDTIME;
                break;
        case 5:
                fintime =+ ORTIME;
                break;
        case 6:
                fintime =+ XORTIME;
                break;
        }
}
/*                      ***** MAXTIME *****

        arguments:      address of first data field of unit,
                        number of fields within unit
        returns:        the maximum completion time of all
                        fields in the unit
        notes:          finds maximum completion time of all
                        fields in the unit
*/
maxtime(cbase, fnum)
char *cbase; int fnum;
{       register int i;
        long *addr;
        char *ca;

        ca = cbase + CLOCK;
        mtime = lz;
        for ( i=1; i <= fnum;  i++)
        {
                if(mtime < *(addr = ca) ) mtime = *(addr = ca);
                ca =+ FLDSIZE * 2;
        }
}
/*                      ***** OTHERQ *****

        arguments:      none
        returns:        none
        notes:          enters other instructions on event queue
                        as specified by instruction
*/
otherq()
{
      long *laddr;
     int i;
     char *cbase;

        cbase = fldaddr(opnd1) ;
        laddr = cbase + CLOCK;
        switch(opspec)
        {
        case 0:
        case 1:
                if(faultbit)
                {
                        fintime = *laddr;
                        qevent(fintime,optype,opspec,cbase,0,opnd2,lz,0.0);
                }
                else
                {
                        fintime=max(*laddr, clock);
                        if(fintime <= clock + DECODETIME)
                                fintime =+ DECODETIME;
                        fintime =+ IOTIME;
                        *laddr = fintime;
                        for( i=1; i<=3; i++)
                        {
                                laddr = cbase + CLOCK + (i * FLDSIZE * 2);
                                *laddr = fintime;
                        }
                        qevent(fintime,optype,opspec,cbase,0,opnd2,lz,0.0);
                }
                break;
        case 2:  /* BOMB */
                printf("nnnnnnnnPROGRAM BOMBED!!!nnnn");
                printf("clock value = %snn",locv(clock) );
                exit();
                break;
        }
}
/*

                    ***** FAULTQ *****

        arguments:      none
        returns:        none
        notes:          enters a special fault injection
                        instruction on the event queue.  Completion
                        time is the last calculated time of the
                        field
*/

faultq()
{   long *addr;
    int *iptr;
    char *cbase, *ca;
    long ltemp;
    float x,y,
          fr,   /* value of frequency */
          *pfr; /* pointer to fr */

        cbase = fldaddr(opnd1);
        fintime = *(addr = ca = cbase + CLOCK);
        switch(opspec)
        {
        case 0:  /* RAP */
                qevent(fintime,optype,opspec,cbase,0,0,lz,0.0);
                break;
        case 1:  /* DE */
                pa =+ 2;
                qevent(fintime,optype,opspec,cbase,0,*pa,lz,0.0);
                break;
        case 2:
        case 3:
                pa =+ 2;
                iptr = &ltemp;
                *iptr++ = *pa;
                *iptr = *(pa + 1);
                qevent(fintime,optype,opspec,cbase,0,0,ltemp,0.0);
                break;
        case 4:
        case 5:   /* RSA1 */
                pfr = &fr;
                pa =+ 2;
                x = *(pa + 2);
                y = *(pa + 3);
                *pfr = x / y;
                iptr = &ltemp;
                *iptr++ = *pa;
                *iptr = *(pa + 1);
                qevent(fintime,optype,opspec,cbase,0,0,ltemp,*pfr);
                pa =+ 2;
                break;
        case 6:  /* RF0 */
        case 7:  /* RF1 */
                pa =+ 2;
                iptr = &ltemp;
                *iptr++ = *pa;
                *iptr = *(pa + 1);
                qevent(fintime,optype,opspec,cbase,0,0,ltemp,0.0);
                break;
        }
}
/*
                    ***** XEVENTS *****

        arguments:      none
        returns:        none
        notes:          while the times of the events in the event
                        queue are less than or equal to the clock,
                        the events are executed as specified
*/
xevents(xtime)
long xtime;
{
    struct node *xptr;  /* pointer to node being executed */
        p = head->next;
        while( p->time <= xtime && p->next != NULL)
        {
                xptr = p;
                p = p->next;
                remove(xptr);
                switch(xptr->type)
                {
                case 0:
                        twox(xptr);
                        break;
                case 1:
                        unitx(xptr);
                        break;
                case 2:
                        onex(xptr);
                        break;
                case 3:
                        imox(xptr);
                        break;
                case 4:
                        twox(xptr);
                        break;
                case 5:
                        branchx(xptr);
                        break;
                case 6:
                        otherx(xptr);
                        break;
                case 7:
                        faultx(xptr);
                        break;
                }
        }
}
/*                      ***** REMOVE *****

        arguments:      pointer to node on the queue
        returns:        none
        notes:          removes node pointed to by qptr and
                        places it on the array nodes
*/
remove(qptr)
struct node *qptr;
{    struct node
        *prevptr,
        *nextptr;

        prevptr = qptr->prev;
        nextptr = qptr->next;
        prevptr->next = nextptr;
        nextptr->prev = prevptr;
        qptr->prev = NULL;
        qptr->next = NULL;
        free(qptr);
}
/*                      ***** MAX *****

        arguments:      two positive integers
        returns:        the larger of the two integers
        notes:          compares the integers and returns the
                        larger of the two

*/

max( i1, i2)
long i1, i2;
{
        if( i1 <= i2 ). return(i2);
        return(i1);
}
/*                      ***** IMMX *****

        arguments:      pointer to node of event to be executed
        returns:        none
        notes:          executes event (immediate.operand
                        instruction) on node pointed to by
                        xptr
*/

immx(xptr)
struct node *xptr;
{
    long *laddr, lval;
    int *iptr;
        laddr = xptr->add1;
        lval = xptr->val;
        switch(xptr->spec)
        {
```

```
        case 0:
                *laddr =+ lval;
                break;
        case 1:
                *laddr =- lval;
                break;
        case 2:
                *laddr =* lval;
                break;
        case 3:
                *laddr =/ lval;
                break;
        case 4:
                *laddr =& lval;
                break;
        case 5:
                *laddr =| lval;
                break;
        case 6:
                iptr = laddr;
                iptr[1] =¬ xptr->val;
                break;
        case 7:
                *laddr = xptr->val;
                break;
        case 8:
                /* compare */
                injfault(xptr->add1);
                if (*laddr == lval) condcode=01;
                else
                        if (*laddr < lval) condcode=02;
                else
                        if (*laddr > lval) condcode=04;
                break;
        case 9:
                /* logical left */
                *laddr =* 2 * xptr->val;
                break;
        case 10:
                /* logical right */
                *laddr =/ (2 * xptr->val);
                break;
        case 11:
                /*arith left */
                *laddr =<< xptr->val;
                break;
        case 12:
                /* arith right */
                *laddr =>> xptr->val;
                break;
        }
        injfault(xptr->add1);
}
/*
                ***** TWOX *****

        arguments:      pointer to node of event to be executed
        returns:        none
        notes:          executes event (two operand instruction)
                        on node pointed to by xptr
*/

twox(xptr)
struct node *xptr;
{
    int *iptr1, *iptr2;
```

```
     long *laddr1, *laddr2;
         laddr1 = xptr->add1;
         laddr2 = xptr->add2;
         switch(xptr->spec)
         {
         case 0:
                 *laddr1 =+ *laddr2;
                 break;
         case 1:
                 *laddr1 =- *laddr2;
                 break;
         case 2:
                 *laddr1 =* *laddr2;
                 break;
         case 3:
                 *laddr1 =/ *laddr2;
                 break;
         case 4:
                 *laddr1 =& *laddr2;
                 break;
         case 5:
                 *laddr1 =| *laddr2;
                 break;
         case 6:
                 iptr1 = laddr1;
                 iptr2 = laddr2;
                 *iptr1 =¬ *iptr2;
                 iptr1[1] =¬ iptr2[1];
                 break;
         case 7:
                 *laddr1 = *laddr2;
                 break;
         case 8:
                 /* comparison */
                 injfault(xptr->add1);
                 intcmpr(*laddr1, *laddr2);
                 break;
         case 9:  /* LDM */
                 *laddr1 = *laddr2;
                 break;
         case 10:  /* STM */
                 injfault(xptr->add1);
                 *laddr2 = *laddr1;
                 break;
         }
         injfault(xptr->add1);
}
/*
                        ***** ONEX *****

         arguments:      pointer to node of event to be executed
         returns:        none
         notes:          executes event (one operand instruction)
                         on node pointed to by xptr
*/
onex(xptr)
struct node *xptr;
{
     long *laddr;
         laddr = xptr->add1;
         switch(xptr->spec)
         {
         case 0:
                 *laddr = -(*laddr);
                 break;
         case 1:
                *laddr = °(*laddr);
                break;
        case 2:
                *laddr = sqrt(*laddr);
                break;
        case 3:
                *laddr = abs(*laddr);
                break;
        case 4:
                *laddr = 0;
                break;
        case 5:
                /* parity check */
                injfault(xptr->add1);
                condcode = getpar(*laddr) << 4;
                break;
        }
        injfault(xptr->add1);
*
                    ***** GETPAR *****

        arguments:      an integer
        returns:        1 if integer of odd parity
                        0 if integer of even parity
        notes:          finds and returns parity of the integer
*/
etpar(numb)
ong numb;
{   int i, par;
        for (par = 0, i=1; i; i=i*2)
                if( i&numb) par++;
        return( par & 01);
}
/*
                    ***** BRANCHX *****

        arguments:      pointer to node of event to be executed
        returns:        none
        notes:          executes event (branch operand instruction)
                        on node pointed to by xptr
*/
branchx(xptr)
struct node *xptr;
{
    int *iptr;
        switch(xptr->spec)
        {
        case 0:  /* branch unconditionally */
        case 1:  /* indirect */
                pa = progbeg + xptr->val;
                break;
        case 2:  /* conditionally */
        case 3:  /* indirect */
                iptr = &xptr->mask;
                if (condcode & iptr[1])
                        pa = progbeg + xptr->val;
                break;
        case 4:  /* branch and save */
        case 5:  /* indirect */
                pa = progbeg + xptr->val;
                regsave[rscount++] = pa + 2;
                break;
        case 6:  /* HLT */
                pa = progend;
                printf("\n\n\nProgram terminated normally\n\n\n");
                printf("Clock = %s\n",locv(clock));
```

```
                    break;
          case 8:  /* RET  */
                    pa = regsave[--rscount];
                    break;
          }
}
/*

                    ***** FAULTX *****

          arguments:        pointer to node of the event to be
                            executed
          returns:          none
          notes:            executes event (special fault injection
                            instruction) on node pointed to by xptr
*/

faultx(xptr)
struct node *xptr;
{
     long *laddr;
     int *iaddr;
     char *base;
     float *flptr;
          base = xptr->add1;
          switch(xptr->spec)
          {
          case 0:  /* RAF */
                    *(iaddr = base + ERRTYPE) = 0;
                    /* assign zeroes to s-a-1 mask, ones to
                              s-a-0 mask  */
                    *(laddr = base + PSA1) = lz;
                    *(laddr = base + RSA1) = lz;
                    *(iaddr = base + PSA0) = 0177777;
                    *++iaddr = 0177777;
                    *(iaddr = base + RSA0) = 01777777;
                    *++iaddr = 0177777;
                    break;
          case 1:  /* DE */
                    movenode(xptr) ;
                    break;
          case 2:  /* SA0 */
                    *(iaddr = base + ERRTYPE) =| 01;
                    if ( *(laddr = base + PSA0) == 0)
                              *laddr = -1;/*initialize all bits to 1*/
                    *laddr =& xptr->mask;
                    break;
          case 3:  /* SA1 */
                    *(iaddr = base + ERRTYPE) =| 02;
                    *(laddr = base + PSA1) =| xptr->mask;
                    break;
          case 4:  /* RSA0 */
                    *(iaddr = base + ERRTYPE) =| 04;
                    if( *(laddr = base + RSA0) == 0)
                              *laddr = -1;/*initialize all bits to 1*/
                    *laddr =& xptr->mask;
                    flptr = base + RSA0FREQ;
                    *flptr = xptr->freq;
                    break;
          case 5:  /* RSA1 */
                    *(iaddr = base + ERRTYPE) =| 08;
                    *(laddr = base + RSA1) =| xptr->mask;
                    flptr = base + RSA1FREQ;
                    *flptr = xptr->freq;
                    break;
          case 6:  /* RFO */
                    if( (temp = *(iaddr = base + ERRTYPE) == 0))
```

```
               return;  /* no faults */
          if ( (temp & 012) == 0) return; /* no SA0 */
          *(laddr = base + PSA0) =| xptr->mask;
          if( *(iaddr = base + PSA0) == 0177777 &&
               *(iaddr = base + PSA0 + 1)==0177777)/*no more SA0*/
                    *(iaddr = base + ERRTYPE) =& 177776;
          *(laddr = base + RSA0) =| xptr->mask;
          if( *(iaddr = base + RSA0) == 0177777 &&
               *(iaddr = base + RSA0 + 1) == 0177777)
                    *(iaddr = base + ERRTYPE) =&-177773;
          break;
     case 7:  /* RF1 */
          if( (temp = *(iaddr = base + ERRTYPE)) == 0)
               return;  /* no faults */
          if( (temp & 05) == 0) return;  /* no SA1 */
          *(laddr = base + PSA1) =& xptr->mask;
          if( *(laddr = base + PSA1) == 0)
               *(iaddr = base + ERRTYPE) =& 0177775;
          if( *(laddr = base + RSA1) == 0)
               *(iaddr = base + ERRTYPE) =& 0177737;
          break;
     }
     injfault(xptr->add1);
}
/*
               ***** MOVENODE *****

     arguments:          old event time, data address, new event time
     returns:            none
     notes:              finds event on event queue corresponding to
                         old event time and data address, and
                         moves if to new event time position on the
                         queue
*/
movenode(xptr)
struct node *xptr;
{
   long oet;  /* old event time */
   long *lptr;
   struct node *pt, *pt2, *saveptr;
     pt = head->next;
     oet = *( lptr = xptr->add1 + CLOCK);
     *lptr =+ xptr->val;
     while( pt->time != oet || pt->add1 != xptr->add1)
     {         if(pt->next == NULL)
                    /* field is not on event queue */
                    return;
               pt = pt->next;
     }
     saveptr = pt;
     saveptr->time = *lptr;
     pt = saveptr->prev;
     pt2 = saveptr->next;
     pt->next = pt2;
     pt2->prev = pt;
     while( pt->time <= saveptr->time) pt = pt->next;
     insert(pt, saveptr);
}
/*
               ***** UNITX *****

     arguments:          pointer to node of event to be executed
     returns:            none
     notes:              executes event (unit operand instruction)
                         on node pointed to by xptr
*/
```

```
unitx(xptr)
struct node *xptr;
{       long o1, o2;
        long *lptr1, *lptr2;
        long *rptr;  /* pointer to result */
        int *iptr1, *iptr2;
        if( xptr->spec <= 6)  /* instruction for alu only */
        {
                rptr = xptr->add1 + 3 * FLDSIZE *2;
                o1 = *( lptr1 = xptr->add1 );
                o2 = *( lptr2 = xptr->add1 + FLDSIZE * 2 );
                switch( xptr->spec)
                {
                case 0:
                        *rptr = o1 + o2;
                        break;
                case 1:
                        *rptr = o1 - o2;
                        break;
                case 2:
                        *rptr = o1 * o2;
                        break;
                case 3:
                        *rptr = o1 / o2;
                        break;
                case 4:
                        *rptr = o1 & o2;
                        break;
                case 5:
                        *rptr = o1 | o2;
                        break;
                case 6:
                        iptr1 = lptr1;
                        iptr2 = lptr2;
                        *iptr1 =¬ *iptr2;
                        iptr1[ 1] =¬ iptr2[ 1 ];
                        break;
                }
        }
        else
        {       switch(xptr->spec)
                {
                case 7:
                        break;
                case 8:
                        break;
                case 9:
                        dumpmem(xptr->add1);
                        break;
                case 10:
                        vote(xptr->add1);
                        break;
                case 11:
                        *(xptr->add1 + STAT) = 1;
                        break;
                case 12:
                        *(xptr->add1 + STAT) = 0;
                        break;
                }
        }
}
/*                      ***** VOTE *****

        arguments:      address of first register of vsd
        returns:        none
        notes:          compares the fields of the vsd for a
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
                                majority value which is placed at the
                                output. If no majority is found,
                                the output register is set and
                                the data output register set to zeroes
*/
vote(cbase)
char *cbase;
{
    int
        numcomp, /* number of registers to compare */
        i, j,
        alike, /* number of other values the field
                                agrees with */
        *iptr;
    long
        *sptr, /* status pointer */
        *lptr1, *lptr2;


        iptr = cbase;
        numcomp = iptr[1];
        if( numcomp <=0 || numcomp >= 9)
        {
                printf("improper mode size, instruction skipped\n\n");
                return;
        }
        sptr = cbase + (11 * FLDSIZE * 2);
        for(i=1; i <= (numcomp/2 + 1); i++)
        {
                lptr1 = cbase + FLDSIZE * 2 * i;
                printf("data%1d=%s",i,locv(*lptr1) );
                printf("addr=%d",lptr1);
                alike = 1;
                for( j=i+1; j<=numcomp; j++)
                {
                        if(*lptr1 == *(lptr2=cbase + FLDSIZE * 2 * j))
                                alike++;
                }
                if(alike > numcomp/2) /* majority */
                {
                        *sptr = lz;
                        lptr2 = cbase + (10 * FLDSIZE * 2);
                        *lptr2 = *lptr1;
                        printf("result = %s", locv(*lptr2));
                        printf("adresult=%d", lptr2);
                        printf("maj alike val=%s\n",locv(*lptr1) );
                        return;
                }
        }
        lptr2 = cbase + (10 * FLDSIZE * 2);
        *lptr2 = lz;
        *sptr = longone;
}
/*                      ***** DUMPMEM *****

        arguments:      address of first register in memory
        returns:        none
        notes:          dumps the entire memory
*/
dumpmem(addr)
int *addr;
{
        dummy = 0;
}
/*
                        ***** OTHERM *****
        arguments:       pointer to node of event to be executed
        returns:         none
        notes:           executes event (two operand instruction)
                         on node pointed to by xptr
*/
otherx(xptr)
struct node *xptr;
{
        char *chase;
        long *laddr;

        laddr = xptr->add1 + FLDSIZE * 2 * 3;
        switch(xptr->spec)
        {
        case 0:
        case 1:
                dummy= 1;
                *laddr = dummy;
                break;
        }
}
/*                      ***** INTCMPR *****

        arguments:       two integers, x and y
        returns:         none

        notes:           sets condition codes comparing x and y,
                         i.e. x = y, x > y, x < y
*/
intcmpr(x, y)
int x,y;
{
        if (x == y) { condcode = 01;  return;}
        if ( x < y ) { condcode = 02;  return;}
        if (x > y) { condcode = 04;  return; }
}



/*                      ***** INITFAULTS *****

        arguments:       none
        returns:         none
        notes:           gets name of fault initialization file,
                         encodes the instruction (as in
                         Phase-II) and decodes the instruction
                         and places on event queue
*/
initfaults()
{    int n;
    int space[6];
    int i;
        printf("Expecting fault initialization file\n");
        openfile(&faultfile);
        initialization = 1;
        while( getline() != -1)
        {
                for (i=0;i<=5;i++) space[i] = 0;
                progaddr = &space[0];
                if(spaces() <= 0)
                {
                        error(18);
                        continue;
                }
                fltime = lz;
                while( *lineptr >= '0' && *lineptr <= '9')
                {
                        fltime =* 10;
                        fltime =+ *lineptr++ - '0';
                }
                if( spaces() <= 0)
                {       error(18);
                        continue;
                }
                if( getname(name) <= 0)
                {       error(22);
                        continue;
                }
                if( name[0] != '*')
                {       error(23);
                        continue;
                }
                codefltinj();
                pa = &space[0];
                ginst();
        }
        initialization = 0;
}


/*                      ***** INJFAULT *****

        arguments:      address of data field
        returns:        none
        notes:          checks whether a field is to
                        have any permanent faults injected
                        and injects the specified faults
*/
injfault(cbase)
char *cbase;
{   long *mskptr, *lptr;
    int *iptr;
    float randnum, *freqptr;

        if( *(iptr = cbase + ERRTYPE) != 0) /*any faults present?*/
        {       lptr = cbase;
                if ( *iptr & 01)  /* SA0 */
                {       mskptr = cbase + PSA0;
                        *lptr =& *mskptr;
                }
                if ( *iptr & 02)  /* SA1 */
                {       mskptr = cbase + PSA1;
                        *lptr =& *mskptr;
                }
                if (*iptr & 04)  /* RSA0 */
                {
                        randnum = rand();
                        randnum =/ 32767;
                        freqptr = cbase + PSA0FREQ;
                        if (randnum <= *freqptr) /* inject RSA0 */
                        {
                                mskptr = cbase + RSA0;
                                *lptr =& *mskptr;
                        }
                }
                if (*iptr & 08)
                {
                        randnum = rand();
                        randnum =/ 32767;
                        printf("in case 08 randnum=%f",randnum);
                        freqptr = cbase + RSA1FREQ;
                        printf("*freqptr=%f\n",*freqptr);
                        if (randnum <= *freqptr) /* inject RSA1 */
                        {
                                mskptr = cbase + RSA1;
                                printf("msk=%s",locv(*mskptr) );
                                *lptr =| *mskptr;
                        }
                }
        }
}
```